全国高等学校计算机教育研究会教材建设立项项目

微机原理与接口技术

——嵌入式系统描述

姚琳 万亚东 汪红兵 编著

清华大学出版社

VISIT…

LANZAROTE
Caliente.COM

微机原理与接口技术

——嵌入式系统描述

姚 琳 万亚东 汪红兵 编著

清华大学出版社
北 京

内 容 简 介

本书内容全面、重点明确、表述简洁,注重将微机接口控制器的基本原理和实际操作相结合,突出软硬件设计中的计算思维模式。全书共 12 章,内容包括微机原理及基本概念、Cortex-M3 处理器体系结构、ARM 汇编、嵌入式系统开发基础、GPIO 控制器、NVIC 及 EXTI 中断控制器、定时器、USART 总线、IIC 总线、SPI 总线、ADC 以及低功耗控制,并配套基于 STM32L15x 系列的实验教程。

本书适合作为非计算机专业微机原理及接口技术的教材,也可作为计算机类嵌入式系统课程的参考教材。

本书封面贴有清华大学出版社防伪标签,无标签者不得销售。
版权所有,侵权必究。侵权举报电话:010-62782989　13701121933

图书在版编目(CIP)数据

微机原理与接口技术:嵌入式系统描述/姚琳,万亚东,汪红兵编著. —北京:清华大学出版社,2019
ISBN 978-7-302-52859-3

Ⅰ. ①微…　Ⅱ. ①姚…　②万…　③汪…　Ⅲ. ①微型计算机–理论　②微型计算机–接口技术
Ⅳ. ①TP36

中国版本图书馆 CIP 数据核字(2019)第 082374 号

责任编辑:谢　琛
封面设计:常雪影
责任校对:焦丽丽
责任印制:沈　露

出版发行:清华大学出版社
　　网　址:http://www.tup.com.cn, http://www.wqbook.com
　　地　址:北京清华大学学研大厦 A 座　　邮　编:100084
　　社 总 机:010-62770175　　邮　购:010-62786544
　　投稿与读者服务:010-62776969, c-service@tup.tsinghua.edu.cn
　　质量反馈:010-62772015, zhiliang@tup.tsinghua.edu.cn
　　课件下载:http://www.tup.com.cn,010-62795954
印 装 者:三河市君旺印务有限公司
经　销:全国新华书店
开　本:185mm×260mm　　印　张:25.5　　字　数:589 千字
版　次:2019 年 8 月第 1 版　　印　次:2019 年 8 月第1 次印刷
定　价:69.00 元

产品编号:083063-01

前言

随着智能制造、物联网、大数据技术的推进和应用,以及新工科建设的需求,数据的采集和感知成为这些技术应用不可或缺的重要环节,各种物联网大赛、创新创业大赛都对软硬件系统设计能力提出了很高的要求,需要学生具有数据感知、处理、传输和分析的综合能力;此外,随着计算思维在计算机基础教学方面的不断推进,思维能力培养已成为教育教学界的共识,计算机硬件系统结构中包含大量计算思维的知识点,如 RISC、CISC、哈佛体系结构、Cache 分层存储、中断处理及优化机制、流水线、串行并行总线技术等,是计算思维培养非常有效的一门课程。微机原理与接口技术是非计算机专业计算机硬件教育的重要课程,本教材以嵌入式系统为对象,对微机的基本原理、ARM 微处理器的接口技术进行梳理,结合大量实验培养学生计算机硬件素养和计算思维能力,提高学生在计算机软硬件系统设计、调试和创新方面的能力,适用于本科非计算机专业学生。

本书选用 Cortex-M3 处理器内核的 STM32L152 系列低功耗微控制器对 ARM 嵌入式系统的体系结构进行讲述,教材以计算机硬件体系涉及的计算思维为主线,第 1 章阐述微型计算机的基本概念、内部架构和嵌入式系统概念;第 2 章以 ARM Cortex-M3 的处理器工作模式、流水、中断等为案例具体阐述硬件设计的方法;第 3 章介绍汇编指令编码、寻址技术并对启动代码进行了分析;第 4 章简述了嵌入式开发流程及 C 语言基础;第 5 ~ 11 章对常用外围控制器 GPIO、EXTI、Timer、USART、IIC、SPI、ADC 的一般性工作原理、STM32L1 系列处理器的具体实现和特色、寄存器级别和库函数级别两个层次的程序设计方法进行了详细阐述;第 12 章对低功耗设计进行了介绍。教材内容兼顾嵌入式处理器及外围控制器原理讲解和应用程序设计,让读者理解 Cortex-M3 处理器的特性,各种控制器的工作原理及使用方法,理解嵌入式处理器架构。

本教材目标定位为软硬件协同设计思维,而不仅仅是会使用和开发嵌入式系统,结合实验设计,让学生必须理解 ARM 架构、外围控制器的工作原理和设计思路,能够进行应用系统设计。

本书适用于工科非计算机专业微机接口技术、嵌入式系统课程,也可作为计算机专业嵌入式开发课程的教材。

作　者

2019 年 4 月

目录

第 1 章　微型计算机与嵌入式系统概论 …… 1

1.1　微型计算机概述 …… 1
1.1.1　微型计算机系统的组成 …… 1
1.1.2　微处理器的发展 …… 3
1.2　微型计算机的基本原理 …… 7
1.2.1　冯·诺依曼体系结构 …… 7
1.2.2　微机的总线 …… 8
1.2.3　哈佛体系结构 …… 14
1.2.4　微处理器的内部结构 …… 15
1.2.5　I/O 接口技术 …… 19
1.2.6　存储器 …… 20
1.2.7　程序的执行过程 …… 24
1.3　嵌入式系统概述 …… 25
1.4　嵌入式系统架构 …… 27
1.5　嵌入式系统的典型应用 …… 28
1.6　典型嵌入式开源硬件和软件系统 …… 31
1.6.1　开源硬件平台 …… 31
1.6.2　嵌入式开源操作系统 …… 33

第 2 章　Cortex-M3 微处理器的体系结构 …… 35

2.1　ARM 微处理器系列介绍 …… 35
2.2　ARM Cortex-M3 体系结构 …… 37
2.2.1　总体架构 …… 37
2.2.2　操作模式 …… 39
2.2.3　寄存器 …… 40
2.2.4　总线 …… 44
2.2.5　存储器 …… 45
2.2.6　中断 …… 50

2.3 STM32L152RET6 微处理器介绍 …… 50
2.4 STM32L152RET6 微处理器的系统结构 …… 51
2.5 STM32L152RET6 微处理器的引脚说明 …… 54
2.6 STM32L152RET6 微处理器的复位和时钟控制 …… 57
2.7 STM32L152RET6 微处理器的存储映射 …… 59

第 3 章 Cortex-M3 处理器的指令系统 …… 63

3.1 Cortex-M3 处理器的指令系统概述 …… 63
3.1.1 指令系统基本概念 …… 63
3.1.2 指令格式 …… 65
3.1.3 寻址方式 …… 67
3.1.4 数据传送指令 …… 68
3.1.5 存储器访问指令 …… 69
3.1.6 算术运算指令 …… 74
3.1.7 逻辑运算指令 …… 77
3.1.8 移位和循环指令 …… 78
3.1.9 比较指令 …… 79
3.1.10 分支控制指令 …… 80
3.1.11 其他指令 …… 81
3.2 ARM 汇编器中的伪指令 …… 83
3.2.1 Thumb 伪指令 …… 83
3.2.2 符号定义伪指令 …… 84
3.2.3 数据定义伪指令 …… 85
3.2.4 汇编控制伪指令 …… 86
3.2.5 其他常用的伪指令 …… 87
3.3 汇编语言的程序结构 …… 88

第 4 章 开发板硬件系统及开发环境 …… 92

4.1 最小系统设计 …… 92
4.2 开发板电路原理图 …… 93
4.2.1 电源 …… 93
4.2.2 复位和启动电路 …… 95
4.2.3 时钟 …… 95
4.2.4 调试接口 …… 96
4.2.5 按键 …… 97
4.2.6 LED 灯 …… 97
4.2.7 显示屏 …… 98
4.2.8 扩展 I/O 口 …… 98

4.3 软件开发环境 …… 99
4.3.1 嵌入式软件开发流程 …… 99
4.3.2 程序开发库 CMSIS …… 101
4.3.3 STM32L52 嵌入式程序开发预备知识 …… 103

第5章 通用输入输出 …… 112

5.1 GPIO 原理 …… 112
5.1.1 GPIO 功能 …… 112
5.1.2 I/O 模式配置 …… 113
5.2 GPIO 寄存器 …… 115
5.3 GPIO 操作函数库 …… 120
5.4 GPIO 实例 …… 128
5.4.1 GPIO 寄存器基本操作 …… 128
5.4.2 GPIO LED 灯控制 …… 130
5.4.3 GPIO 按键输入 …… 131

第6章 异常和中断处理技术 …… 133

6.1 中断的基本概念 …… 133
6.2 中断向量表 …… 134
6.3 中断的执行过程 …… 138
6.3.1 中断响应基本流程 …… 138
6.3.2 中断优化技术 …… 141
6.3.3 系统异常 …… 142
6.4 嵌套向量中断控制器 NVIC …… 144
6.4.1 STM32L152 NVIC …… 144
6.4.2 NVIC 寄存器 …… 144
6.4.3 系统异常处理 …… 149
6.4.4 全局中断管理 …… 150
6.4.5 NVIC 库函数 …… 151
6.5 外部中断/事件控制器 EXTI …… 157
6.6 寄存器说明 …… 158
6.7 EXTI 函数库 …… 162
6.8 中断案例 …… 164

第7章 定时器 …… 168

7.1 定时器原理概述 …… 168
7.2 内部定时器 SysTick …… 170
7.2.1 SysTick 寄存器 …… 170

7.2.2 SysTick 定时器库函数 …… 172
7.2.3 SysTick 定时器应用例程 …… 174
7.3 外围定时器基本概念 …… 175
7.4 基本定时器 TIM6、TIM7 …… 179
7.5 通用定时器 TIM2 ~ TIM4、TIM9 ~ TIM11 …… 181
7.5.1 通用定时器时基单元 …… 182
7.5.2 通用定时器输入捕获和输出比较单元 …… 185
7.5.3 TIMx 的外部触发同步模式 …… 189
7.6 定时器寄存器 …… 191
7.7 外围定时器库函数 …… 197
7.8 定时器应用例程 …… 205
7.8.1 定时器寄存器操作案例 …… 205
7.8.2 基本计时中断示例 …… 206
7.8.3 比较输出示例 …… 208
7.8.4 输入捕获示例 …… 212
7.8.5 PWM 输出和输入示例 …… 215

第 8 章 USART 串口控制器 …… 218

8.1 串行输入输出接口的基本概念 …… 218
8.2 串行通信协议 …… 219
8.2.1 异步串行通信协议 …… 219
8.2.2 同步串行通信协议 …… 220
8.2.3 串行通信基本概念 …… 221
8.3 STM32L152 USART 内部结构与原理 …… 224
8.3.1 发送器 …… 226
8.3.2 接收器 …… 228
8.3.3 校验控制 …… 232
8.3.4 硬件流控制 …… 232
8.3.5 USART 中断请求 …… 233
8.4 USART 寄存器 …… 234
8.5 USART 数据传输配置 …… 240
8.5.1 波特率计算 …… 240
8.5.2 异步双向通信模式配置 …… 241
8.6 USART 帧传输协议 …… 242
8.6.1 串行链路帧格式设计 …… 242
8.6.2 MODBUS 帧格式 …… 246
8.7 USART 函数库 …… 247
8.7.1 寄存器定义 …… 247

8.7.2 USART 库函数 …… 250
8.8 USART 案例 …… 257
8.8.1 串口寄存器操作案例 …… 257
8.8.2 串口配置基本流程 …… 258
8.8.3 PC 串口通信案例 …… 259
8.8.4 状态机多字节数据帧发送和接收案例 …… 261

第 9 章 IIC 总线 …… 267

9.1 IIC 总线概述 …… 267
9.2 I2C 总线的基本操作 …… 268
9.3 STM32L152 I2C 总线控制器 …… 273
9.4 I2C 寄存器描述 …… 275
9.5 I2C 数据通信流程 …… 281
9.5.1 I2C 从模式通信 …… 281
9.5.2 I2C 主模式通信 …… 283
9.5.3 总线通信错误 …… 285
9.5.4 中断请求 …… 285
9.6 函数库 …… 287
9.6.1 I2C 寄存器结构 …… 287
9.6.2 I2C 库函数 …… 288
9.7 I2C 案例 …… 298
9.7.1 I2C 寄存器操作案例 …… 298
9.7.2 I2C 基本配置 …… 299
9.7.3 模拟 I2C 实现 …… 301
9.7.4 串行 Flash 通信 …… 304
9.7.5 ADT7420 温度传感器通信 …… 306

第 10 章 SPI …… 309

10.1 SPI 总线概述 …… 309
10.2 SPI 总线控制器架构 …… 310
10.2.1 接口信号和连接方式 …… 310
10.2.2 传输模式和时序 …… 313
10.2.3 STM32L15x SPI 总线控制器 …… 315
10.3 SPI 寄存器说明 …… 317
10.4 SPI 通信流程 …… 320
10.4.1 SPI 双工通信模式配置 …… 321
10.4.2 SPI 单工/半双工通信 …… 323
10.5 函数库 …… 326

10.5.1 SPI 寄存器结构 …… 326
10.5.2 SPI 库函数 …… 328
10.6 SPI 案例 …… 333
10.6.1 SPI 寄存器操作案例 …… 333
10.6.2 SPI 函数库案例 …… 334
10.6.3 温度传感器 ADT7320 案例 …… 335

第 11 章 模拟/数字转换 …… 339

11.1 ADC 简介 …… 339
11.2 STM32L152 ADC …… 344
11.2.1 STM32L152 ADC 功能 …… 346
11.2.2 温度和电压转换 …… 357
11.3 ADC 寄存器 …… 358
11.4 ADC 寄存器结构及 ADC 库函数 …… 365
11.4.1 ADC 寄存器结构 …… 366
11.4.2 ADC 库函数 …… 368
11.5 ADC 案例 …… 377
11.5.1 ADC 寄存器操作案例 …… 377
11.5.2 ADC 库函数操作案例 …… 378

第 12 章 低功耗技术 …… 380

12.1 处理器功耗的构成/类型 …… 380
12.1.1 动态功耗 …… 380
12.1.2 静态功耗 …… 381
12.2 STM32L1 系列处理器低功耗设计 …… 382
12.2.1 STM32 的电源系统 …… 382
12.2.2 动态电压调节管理 …… 383
12.2.3 电源检测 …… 385
12.2.4 低功耗模式 …… 386
12.3 功耗控制寄存器 …… 389
12.4 PWR 寄存器结构及库函数 …… 391
12.4.1 PWR 寄存器结构 …… 391
12.4.2 PWR 库函数 …… 391
12.5 PWR 案例 …… 394

参考文献 …… 397

Chapter 1

第1章　微型计算机与嵌入式系统概论

【导读】 嵌入式系统属于微型计算机范畴，本章首先介绍微型计算机的基本组成，核心部件微处理器的发展历史，然后对微处理器工作原理的一些基本概念：组成架构、总线、输入输出、存储系统等进行了介绍。微处理器应用于微机，属于通用计算机系统；而与应用场景结合，形成专用计算系统，称为嵌入式系统，本章对嵌入式系统的概念、组成和典型应用进行介绍，并列举了目前典型的开源嵌入式开发硬件和软件平台。通过本章学习，建立微机系统的整体构成和微处理器设计的相关计算思维方法，对嵌入式系统的概念及软硬件系统有总体的认识。

1.1　微型计算机概述

微型计算机是针对小型计算机、大型计算机和超级计算机而言的，是根据规模和性能进行计算机分类的。一般来说，微型计算机是一种小型的、相对便宜的、以微处理器作为CPU的计算机。这类计算机由印制电路板、微处理器、存储器和输入输出电路组成，占用很少的物理空间。随着集成电路技术的发展，微型计算机在20世纪80年代开始流行，获得广泛的应用。目前我们使用的个人计算机（台式机、笔记本计算机、平板计算机、智能手机、计算器等）、家用娱乐设施（游戏机、智能电视、智能音箱、电子书阅读器等）以及路由器、交换机等通信设备均是微型计算机系统。

1.1.1　微型计算机系统的组成

一个完整的微型计算机系统由硬件系统和软件系统两部分组成，如图1-1所示。

计算机硬件部分包括中央处理器（CPU）、存储器、输入和输出设备。

（1）微型计算机的中央处理器也称为微处理器（Micro Processor Unit，MPU）。计算机利用CPU处理数据，利用存储器存储数据。CPU是计算机硬件的核心，主要包括运算器和控制器两大部分，控制着整个计算机系统的工作。计算机的性能主要取决于CPU的性能。

运算器又称为算术逻辑单元（Arithmetic Logic Unit，ALU），控制器的主要作用是使整个计算机系统能够自动运行。控制器从存储器取出数据，运算器进行算术运算或逻辑运算，

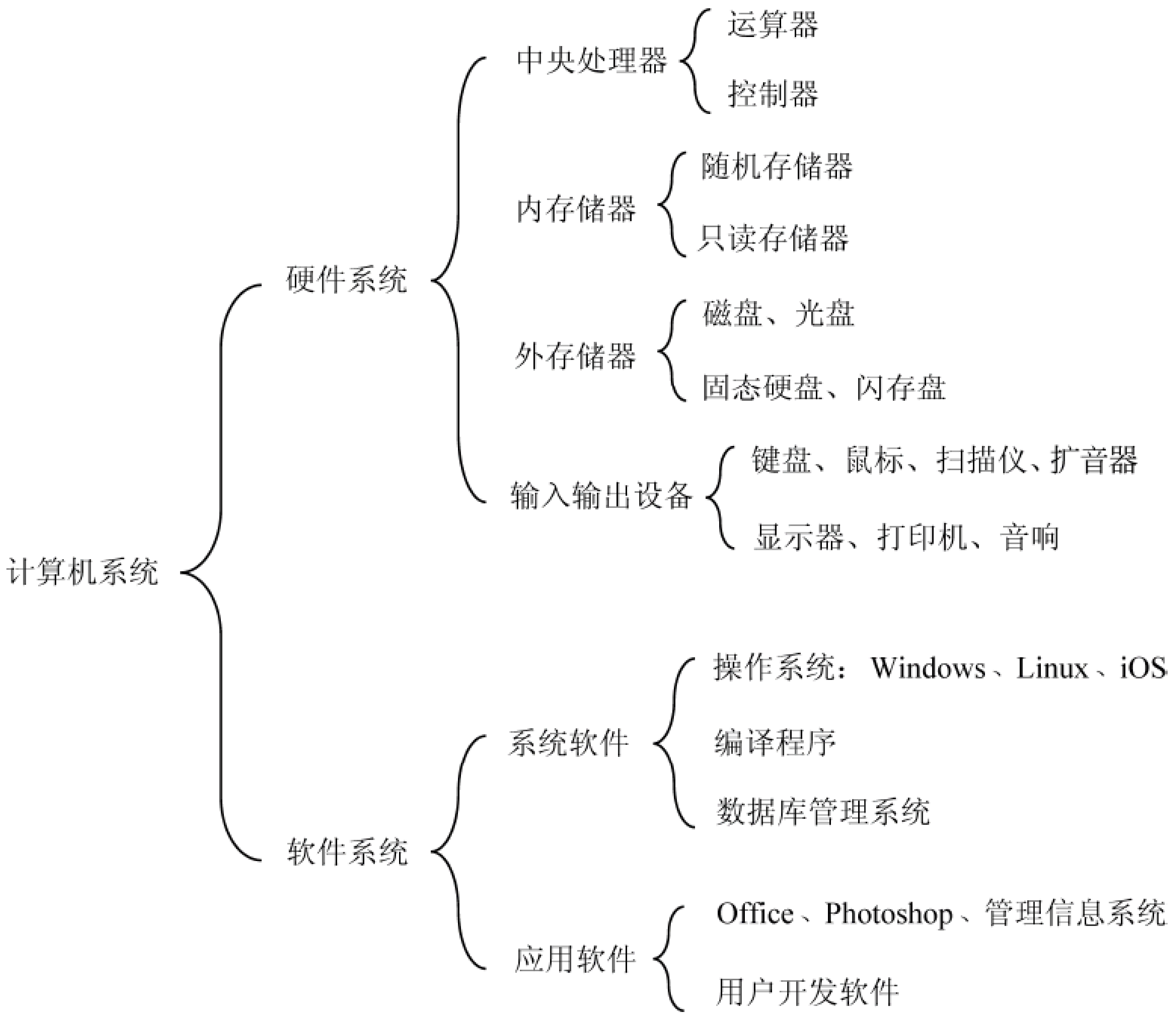


图 1-1　计算机的系统组成

并把处理后的结果送回存储器。执行程序时，控制器从主存中取出相应的指令和数据，然后向其他功能部件发出指令所需的控制信号，完成相应的操作，再从主存中取出下一条指令执行，如此循环，直到程序结束。

（2）存储器是计算机中的存储部件。存储器分为主存（内存储器）和辅存（外存储器）两大类。在计算机系统中，习惯上把内存、CPU 合称为主机。内存储器分为随机读写存储器（RAM）、只读存储器（ROM）和高速缓冲存储器（Cache）三类。通常生活中的内存一般指的是 RAM。外存储器主要包括硬盘、光盘、U 盘和移动硬盘等。

（3）输入输出设备主要包括键盘、鼠标、显示器和打印机等。

硬件是组成计算机的基础，软件是计算机的灵魂。计算机的硬件系统上只有安装了软件后，才能发挥其应有的作用，使用不同的软件，计算机可以完成各种不同的工作。微型计算机系统的软件分为两大类，即系统软件和应用软件。

系统软件是指为管理计算机系统的硬件和支持应用软件运行而提供的基本软件，最常用的有操作系统、程序设计语言编译器、数据库管理系统、联网及通信软件等。操作系统是微机最基本、最重要的系统软件，它负责管理计算机系统的各种硬件资源（例如 CPU、内存空间、磁盘空间、外部设备等），并且负责将用户对机器的管理命令转换为机器内部的实际操作。典型的操作系统有 Linux、Mac OS、Windows 7、Windows 10、iOS 等。

应用软件是指除了系统软件以外，利用计算机为解决某类问题而设计的程序的集合，主要包括信息管理软件、辅助设计软件、实时控制软件等。

1.1.2 微处理器的发展

微型计算机系统的核心是微处理器，微处理器的发展直接影响着微型计算机系统的应用。

1.1.2.1 微处理器发展史

1）早期微处理器

第一款微处理器是美国军方研制的中央空气数据计算机(Center Air Data Computer，CADC)，由 6 颗晶片组成，用于 F-14 雄猫战机的大气数据测量与控制。1971 年，Intel 公司发布的 4004 是世界上第一款商用处理器，主频 108kHz，4004 和 Intel 开发出的 4001(动态内存 DRAM)、4002(只读存储器 ROM)、4003(寄存器 Register)可架构出一台微型计算机硬件系统。

2）8 位微处理器时期

1972 年，Intel 公司推出的 8008 微处理器是第一款 8 位处理器，主频 0.5MHz。1974 年，Intel 公司推出 8080 处理器，主频 2MHz，16 位地址总线、8 位数据总线，内部集成 7 个 8 位寄存器，支持 16 位内存，同时也包含了一些输入输出端口。

此时，微处理器的优势已被业界所认同，更多公司开始进入微处理器设计，仙童、AMD、摩托罗拉以及 Zilog 等公司均开始研发微处理器，摩托罗拉 1974 年发布了 MC6800，工作主频 1MHz，1976 年 Zilog 公司发布了 Z80，性能比 8080 更强大。

3）16 位微处理器时期

1978 年，Intel 公司首次生产出 16 位的微处理器，命名为 8086，同时还生产出与之相配合的数学协处理器 8087，这两种芯片使用相互兼容的指令集，即后来 PC 使用的 X86 指令集。

1979 年，Intel 公司推出了 8088 芯片，主频 4.77MHz，地址总线为 20 位，寻址范围 1MB 内存。8088 内部数据总线都是 16 位，外部数据总线是 8 位(8086 是 16 位)。1981 年 8088 芯片首次用于 IBM PC 中，开创了全新的微机时代。

1979 年，Zilog 发布了其第一款 16 位处理器 Z8000，摩托罗拉发布 16 位处理器 MC68000(16 位计算单元，32 位数据总线)。

1982 年，Intel 公司推出 80286 芯片，它比 8086 和 8088 都有了飞跃式的发展，虽然它仍旧是 16 位结构，时钟频率 20MHz。其内部和外部数据总线皆为 16 位，地址总线 24 位，可寻址 16MB 内存。

4）32 位微处理器时期

世界上第一块单片 32 位微处理器是 1982 年 AT&T(贝尔)实验室的 BELLMAC-32A。

1985 年，Intel 公司发布了 80386，首次在 X86 处理器中实现了 32 位系统，集成 80387 数字辅助处理器增强浮点运算能力，首次采用外置的高速缓存(Cache)解决内存速度瓶颈问题。工作频率也从 12.5MHz 逐步提高到 20MHz、25MHz、33MHz，直至最后的 40MHz。

图 1-2　Intel 早期 4 位、8 位、16 位处理器

1985 年摩托罗拉推出了 MC68020，增加了 32 位数据和地址总线，在 UNIX 超级微机市场上获得巨大成功，后续又生产了 MC68030(集成了内存管理)、MC68040(集成浮点运算器)。

1986 年 MIPS 推出 R2000 处理器，1988 年推出 R3000 处理器，采用精简指令集(RISC)设计，成为 RISC 微处理器的代表。

1987 年 Sun 公司推出了第一款 32 位的 SPARC 86900 Sunrise 处理器，这款处理器采用 SPARC V7 架构，采用 0.8μm 工艺，主频 16MHz，主要用于 SUN 工作站(Solaris 系统)。

1989 年 Intel 公司发布 80486，支持虚拟存储管理技术，虚拟存储空间 64TB(支持 48 位的有效虚拟地址)。片内集成有浮点运算部件和 8KB 的 Cache，同时也支持外部 Cache。整数处理部件采用精简指令集 RISC 结构，提高了指令的执行速度。此外，80486 微处理器还引进了时钟倍频技术和新的内部总线结构，从而使主频可以超出 100MHz。

之后，Intel 公司陆续发布了 Pentium 系列处理器 Pentium(超标量)、Pentium Pro(动态执行)、Pentium Ⅱ(MMX 指令集)、Pentium Ⅲ(SIMD)、Pentium 4、Pentium M(低功耗)、Pentium D(双核)、酷睿 Core 等一系列处理器。目前，酷睿系列(Core M、Core I3、Core I5、Core I7、Core I9)已经经过了 8 代的发展。

1991 年，ARM 发布了自己的第一款 RISC 处理器核 ARM6，1993 年推出 ARM7，一直到目前的 ARM11 和 Cortex，在嵌入式微处理器市场占据了大量份额。

5) 64 位微处理器时期

1991 年，MIPS 推出第一款 64 位商用微处理器 R4000，之后又陆续推出 R8000、R10000 和 R12000 等型号，2007 年，中科院计算所龙芯获得 MIPS 处理器 32 位和 64 位的授权。

1992 年，DEC 发布了 64 位的 Alpha 处理器，主要用于工作站和服务器，后 DEC 将 Alpha 技术出售给康柏公司，最终康柏公司将 Alpha 的技术出售给 Intel 公司。

1995 年，Sun 公司推出了 64 位 UltraSPARC Ⅰ微处理器。UltraSPARC I 革新了微处理器的可扩展性和带宽等工业标准，其频率达 143MHz，采用 0.5μm 工艺技术。

2001 年，Intel 公司推出了 IA64 架构的 Itanium 系列处理器，其指令系统与 X86 不兼容，用于服务器市场。

2003 年，AMD 提出了 X86 的 64 位扩展指令集 AMD64，用于服务器市场的 64 位处理器进入到 PC 领域，后来 Intel 公司最终采用了 AMD64。

ARM 于 2011 年发布了 ARMv8 64 位架构，ARMv8 使用了两种执行模式：AArch32 和 AArch64，处理器在运行中可以无缝地在两种模式间切换。这意味着 64 位指令的解码

器是全新设计的,不用兼顾 32 位指令,而处理器依然可以向后兼容。

1.1.2.2　微控制器发展史

微控制器(Micro Controller Unit,MCU)是一种采用超大规模集成电路技术把具有数据处理能力的微处理器 MPU、随机存储器 RAM、只读存储器 ROM、多种 I/O 接口和中断系统、定时器/计数器等功能集成到一块硅片上构成的一个小而完善的微型计算机系统。

早期的微控制器称作单片机(Single Chip Microcomputer,SCM),主要实现单片系统集成,随着嵌入式应用的扩展,系统各种外围接口电路日趋复杂,系统的智能化控制能力凸显,各种电气、控制和电子技术厂家开始进入微处理器的设计和生产,结合 SoC(System on Chip)技术,单片机进入微控制器阶段。

1976 年 Intel 公司推出了第一款单片机 MCS-48。此后,GI 推出了 PIC1x 系列 8 位单片机,摩托罗拉推出了 MC6800 处理器为核心的 M6800 系列单片机,Zilog 推出了 Z8 系列单片机。

1990 年,Intel 公司推出了 MCS-51,MCS-51 采用经典的 8 位单片机的总线结构,包括 8 位数据总线、16 位地址总线、控制总线及具有很多机通信功能的串行通信接口,并授权给其他厂家使用,Intel、Atmel、Philips 和 STC 等推出了一系列 MCS-51 核心的单片机,从此成为 8 位单片机的主流,直到现在还在大量应用。此外,8 位单片机的主流架构还有 Atmel 公司的 AVR 系列,凌阳科技的 SPMC65 系列等。

8 位单片机占据了较大的市场,16 位单片机由于 8 位单片机的市场地位和 32 位单片机的快速发展,相对发展空间较小,典型的芯片包括 Intel 的 MCS-96 系列单片机,TI 的 MSP430 系列,摩托罗拉的 68HC12/16 系列以及 MicoChip 的 PIC24 系列。

随着嵌入式系统应用的爆发,单片机的运算处理性能和外设通信能力等无法满足复杂应用的计算需求,由此催生了 32 位微控制器。32 位微控制器多采用 RISC 指令集,典型的代表有 MIPS、ARM、摩托罗拉的 68K 系列等。目前,微控制器基本已被 ARM 占据,随着物联网应用的发展,芯片厂家的主要在低功耗、低电压、大容量存储和高性能计算上对微控制器进行优化。

1.1.2.3　CISC 和 RISC 设计方法

按照指令系统分类,计算机大致可以分为两类:复杂指令系统计算机(Complex Instruction Set Computer,CISC)和精简指令系统计算机(Reduced Instruction Set Computer,RISC)。CISC 是 CPU 的传统设计模式,其指令系统的特点是指令数目多而复杂,每条指令的长度不尽相等;而 RISC 则是 CPU 的一种新型设计模式,其指令系统的主要特点是指令条数少且简单,指令长度固定。

1) CISC 指令集

计算机的指令系统最初只有很少的一些基本指令,而其他的复杂指令全靠软件编译时通过简单指令的组合来实现。后来,越来越多的复杂指令被加入到了指令系统中,可用硬件实现复杂的运算。但是,一个指令系统的指令条数受到指令操作码位数的限制,如果操作码

为 8 位，那么指令条数最多为 256 条，而指令的宽度则是很难增加的。操作码扩展可以解决这个问题，在指令格式中，操作码后面跟的是地址码，而有些指令是用不到地址码或只用少量位数的地址码的，那么就可以把操作码扩展到地址码的位置，使操作码的位数得以增加。例如，一个指令系统的操作码为 2 位，那么可以有 00、01、10、11 四条不同的指令。现在把 11 作为保留，把操作码扩展到 4 位，那么就可以有 00、01、10、1100、1101、1110、1111 七条指令，其中 1100、1101、1110、1111 这四条指令的地址码部分必须减少两位。为了达到减少地址码这一操作码扩展的先决条件，设计者提出了各种各样的寻址方式，如基址寻址、相对寻址等，以最大限度地压缩地址码长度，为操作码留出空间。

由此，大量的复杂指令、可变的指令长度、多种寻址方式形成了 CISC 指令集。CISC 指令集的复杂性大大增加了译码的难度，早期计算机运算能力差，译码相对时间较短，现代计算机运算速度大大提升，导致译码上所浪费的时间过长，严重影响了计算机性能。

2）RISC 指令集

1975 年，IBM 对 IBM 370 CISC 系统的研究发现，发现其中仅占总指令数 20％的简单指令却在程序调用中占据了 80％，而占指令数 80％的复杂指令却只有 20％的机会被调用到，即符合经济学中的 80/20 法则，由此提出了 RISC 的概念。20 世纪 80 年代末开始，各家公司的 RISC CPU 大量出现，其中典型代表为 MIPS 和 ARM。

RISC 体系结构的基本思想：针对 CISC 指令系统指令种类太多、指令格式不规范、寻址方式太多的缺点，通过减少指令种类、规范指令格式、简化寻址方式，方便处理器内部的并行处理，提高处理器内部器件的使用效率，从而大幅度地提高处理器的性能。

RISC 的目标决不是简单地缩减指令系统，而是使处理器的结构更简单、更合理，具有更高的性能和执行效率，同时降低处理器的开发成本。由于 RISC 指令系统仅包含最常用的简单指令，因此，RISC 技术可以通过硬件优化设计，把时钟频率提得很高，从而实现整个系统的高性能。同时，RISC 技术在 CPU 芯片上设置大量寄存器，用来把常用的数据保存在这些寄存器中，大大减少对存储器的访问，用高速的寄存器访问取代低速的存储器访问，从而提高系统整体性能。

RISC 的典型特征包括：

（1）指令种类少，指令格式规范：RISC 指令集通常只使用一种或少数几种格式，指令长度单一（一般 4 字节），并且在字边界上对齐，字段位置（特别是操作码的位置）固定。

（2）寻址方式简化：几乎所有指令都使用寄存器寻址方式，其他更为复杂的寻址方式，如间接寻址等，则由软件利用简单的寻址方式来合成。

（3）大量利用寄存器间操作：RISC 指令集中大多数操作都是寄存器到寄存器的操作，只有取数指令、存数指令访问存储器。

（4）简化处理器结构：使用 RISC 指令集，可以大大简化处理器中的控制器和其他功能单元的设计，不必使用大量专用寄存器，特别是允许以硬连线方式来实现指令操作，以期更快的执行速度，而不必像 CISC 处理器那样使用微程序来实现指令操作。因此，RISC 处理器不必像 CISC 处理器那样设置微程序控制存储器，从而能够快速地直接执行指令。

(5) 加强处理器的并行能力：RISC 指令集非常适合于采用流水线、超流水线和超标量技术，从而实现指令级并行操作，提高处理器的性能。目前常用的处理器的内部并行操作技术，基本上都是基于 RISC 体系结构而逐步发展和走向成熟的。

(6) RISC 技术的复杂性在于它的优化编译程序，因此软件系统开发时间比 CISC 机器要长。

RISC 与 CISC 的主要特征对比如表 1-1 所示。

表 1-1　RISC 与 CISC 的主要特征对比

比 较 项 目	CISC	RISC
指令系统	复杂、庞大	简单、精简
指令数目	一般大于 200	一般小于 100
指令格式	一般大于 4 种	一般小于 4 种
寻址方式	一般大于 4 种	一般小于 4 种
指令字长	不固定	定长
指令执行时间	慢	快
程序代码长度	短	长

1.2　微型计算机的基本原理

1.2.1　冯・诺依曼体系结构

现代计算机基本沿用冯・诺依曼体系结构，其基本设计思想包括以下三点。

1) 计算机系统的组成

运算器、存储器(主存)、控制器、输入设备和输出设备五大部件组成一个完整的计算机系统，如图 1-3 所示。

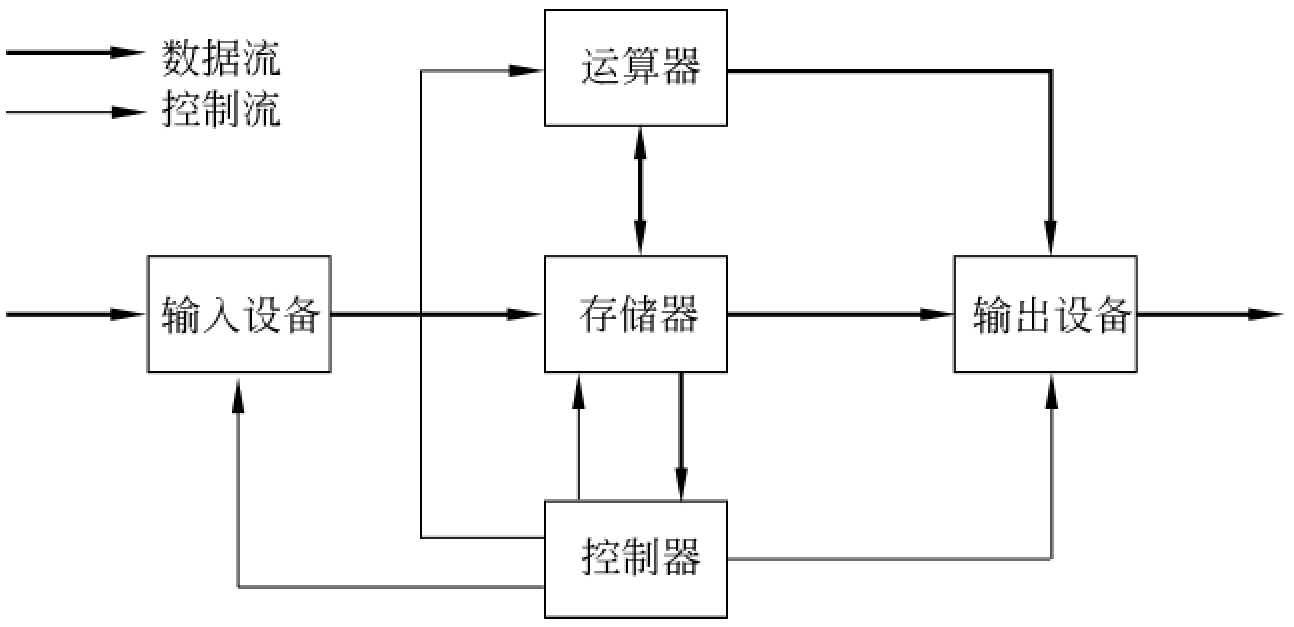


图 1-3　冯・诺依曼体系结构五大功能部件

2）采用二进制形式表示数据和指令

数据和指令都是以二进制形式存储在存储器中，从存储器存储的内容来看两者并无区别，都是由 0 和 1 组成的代码序列，只是各自约定的含义不同。计算机在读取指令时，把从计算机读到的信息看作指令；而在读取数据时，把从计算机读到的信息看作操作数。数据和指令在软件编制中就已加以区分，所以正常情况下两者不会产生混乱。我们通常把存储在存储器中的数据和指令统称为数据，因为程序本身也可以作为被处理的对象进行加工处理，例如对照程序进行编译，就是将源程序当作被加工处理的对象。

3）采用存储程序方式

事先编制程序，将程序(包含指令和数据)存入主存储器中，计算机在运行程序时就能自动地、连续地从存储器中依次取出指令且执行。这是计算机能高速自动运行的基础。计算机的工作体现为执行程序，计算机功能的扩展在很大程度上也体现为所存储程序的扩展。

冯·诺依曼机的这种工作方式，可称为指令流驱动方式，即按照指令的执行序列，依次读取指令，然后根据指令所含的控制信息，调用数据进行处理。因此，在执行程序的过程中，始终以控制信息流为驱动工作的因素，而数据信息流则是被动地被调用处理。为了控制指令序列的执行顺序，设置一个程序(指令)计数器 PC(Program Counter)，让它存放当前指令所在的存储单元的地址。如果程序现在是顺序执行的，每取出一条指令后 PC 内容加 1，指示下一条指令该从何处取得。如果程序将转移到某处，就将转移的目标地址送入 PC，以便按新地址读取后继指令。所以，PC 就像一个指针，一直指示着程序的执行进程，也就是指示控制流的形成。由于多数情况下程序是顺序执行的，所以大多数指令需要依次地紧挨着存放，除了个别即将使用的数据可以紧挨着指令存放外、一般将指令和数据分别存放在该程序区的不同区域内。

冯·诺依曼型计算机从本质上讲是采取串行顺序处理的工作机制，即使有关数据已经准备好，也必须逐条执行指令序列。而提高计算机性能的根本方向之一是并行处理。因此，近年来人们谋求突破传统冯·诺依曼体制的束缚，这种努力被称为非诺依曼化，主要表现在以下三个方面。

- 在冯·诺依曼体制范畴内，对传统冯·诺依曼机进行改造，如采用多个处理部件形成流水处理，依靠时间上的重叠提高处理效率；又如组成阵列机结构，形成单指令流多数据流，提高处理速度。
- 用多个冯·诺依曼机组成多机系统，支持并行算法结构。
- 从根本上改变冯·诺依曼机的控制流驱动方式。例如，采用数据流驱动工作方式的数据流计算机，只要数据已经准备好，有关的指令就可并行执行。这是真正非诺依曼化的计算机，它为并行处理开辟了新的前景，但由于控制的复杂性，仍处于实验探索之中。

1.2.2 微机的总线

1.2.2.1 总线基本概念

冯·诺依曼机的五大功能部件之间是通过总线(Bus)进行连接的。总线是用于连接多

个设备的数据通道，是计算机各种功能部件之间传送信息的公共通信干线，它是由导线组成的传输线束，一根线路在同一时间内仅能传输一比特，因此，必须同时采用多条线路才能传送更多数据，总线可同时传输的数据数就称为总线宽度(Bus Width)，以比特为单位，总线宽度愈大，传输性能就越高。总线频率是总线工作速度的一个重要参数，总线频率是指一秒钟传输数据的次数，工作频率越高，速度越快。总线频率通常用 MHz 表示，如 33MHz、100MHz、400MHz、800MHz 等。总线的带宽(Bandwidth)，即单位时间内可以传输的总数据数，指的是总线本身所能达到的最高数据传输速率。总线带宽＝频率×宽度，单位为 B/s。

总线上传送的信息包括数据信息、地址信息和控制信息，因此，总线按功能分三种：数据总线(Data Bus，DB)、地址总线(Address Bus，AB)和控制总线(Control Bus，CB)，如图 1-4 所示。

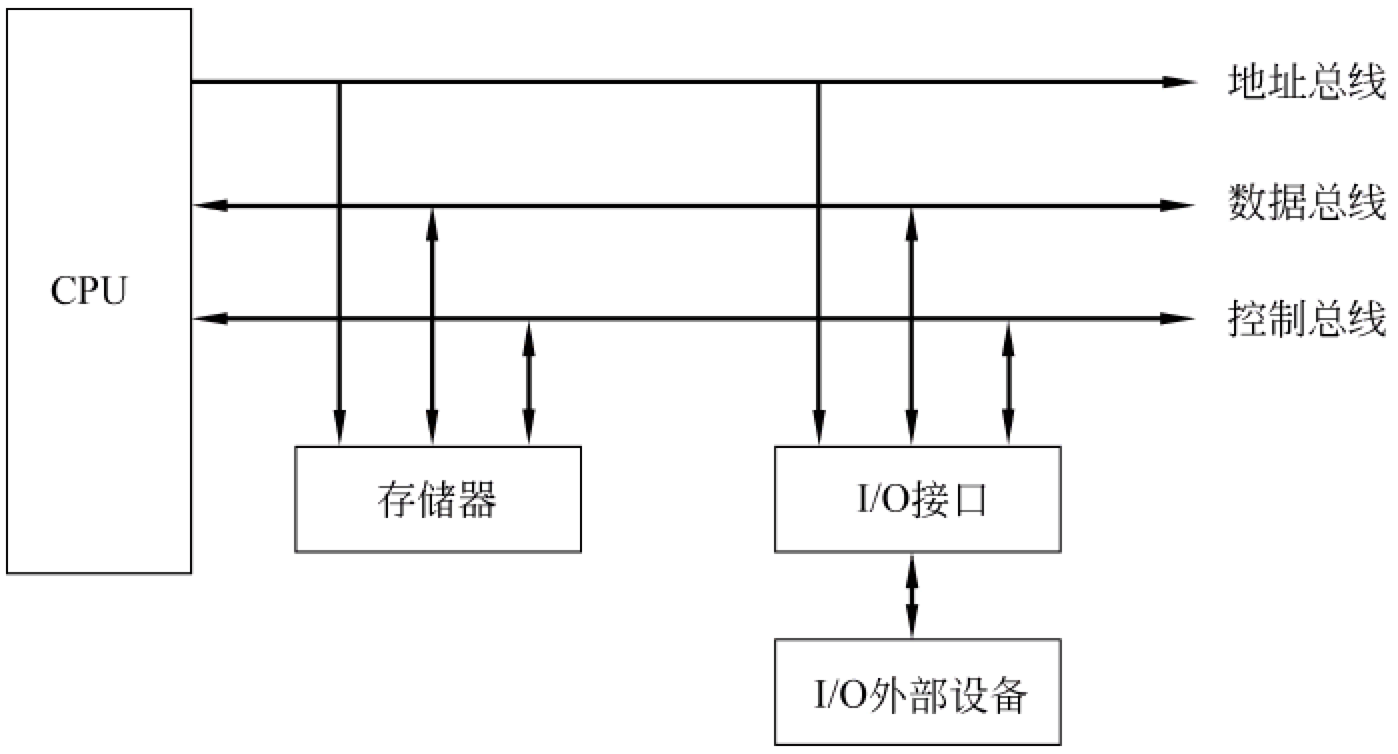


图 1-4　微机系统总线

数据总线用于传送数据信息(包括指令和数据)。数据总线是双向三态总线，既可以把 CPU 的数据信息传送到存储器或 I/O 接口等其他部件，也可以将其他部件的数据信息传送到 CPU。数据总线的位数是微型计算机的一个重要指标，通常与微处理器的字长一致。例如 Intel 8086 微处理器字长 16 位，其数据总线宽度也是 16 位。数据的含义是广义的，它可以是真正运算需要的数据，也可以指令代码或状态信息，有时甚至是一个控制信息，因此，在实际工作中，数据总线上传送的并不一定仅仅是运算数据。

地址总线是专门用来传送地址的，由于地址只能从 CPU 传向外部存储器或 I/O 端口，所以地址总线总是单向三态的。地址总线的位数决定了 CPU 可直接寻址的内存空间大小，比如地址总线为 16 位，则其最大可寻址空间为 $2^{16}=64$KB，地址总线为 20 位，其可寻址空间为 $2^{20}=1$MB。一般来说，若地址总线为 n 位，则可寻址空间为 2^n 个地址空间(存储单元)。例如，我们常用的 32 位处理器多采用 32 位址总线，可以寻址的最大主存空间为 4GB。

控制总线用来传送控制信号和时序信号。控制信号中，有的是微处理器送往存储器和 I/O 接口电路的，如读写信号、片选信号、中断响应信号等；也有其他部件反馈给 CPU 的，如

中断申请信号、复位信号、总线请求信号等。因此，控制总线的传送方向由具体控制信号而定，一般是双向的。控制总线的位数要根据系统的实际控制需要而定，实际上控制总线的具体情况主要取决于 CPU。

1.2.2.2 总线的分类

总线按照功能可以分为数据总线、地址总线和控制总线，按照层次、传输方式等不同的角度可以有不同的分类。

总线在各个层次上提供部件之间的连接和信息交换通路，按不同的层次分为 3 种。

- 内部总线：指芯片内部连接各元件的总线。例如 CPU 芯片内部，在各个寄存器、ALU、指令部件等各元件之间有总线相连。
- 系统总线：指连接 CPU、存储器和各种 I/O 模块等主要部件的总线，又称为板级总线或板间总线。
- 局部总线：处理器-主存专用总线、高速 I/O 总线等，这类总线用于主机和 I/O 设备之间或计算机系统之间的通信。

按总线的数据传输方式可以分为两种。

- 串行总线：在数据线上按位进行传输，只需一根数据线，线路成本低，适合于远距离数据传输。串行总线早期为慢速总线，连接慢速设备，目前已出现了大量高速串行总线，如 USB、1394、DP 等。
- 并行总线：在数据线上同时有多位一起传送，每一位有一根数据线，故需多根数据线，速度比串行总线快，如 ATA、PCI 等。

1.2.2.3 串行和并行传输

1）串行传输

当信息以串行方式传输时，只需要一条传输线，且采用脉冲传送。在串行传送时，按顺序传送表示一个数码的所有二进制位（bit）的脉冲信号，每次一位，通常以第一个脉冲信号表示数码的最低有效位，最后一个脉冲信号表示数码的最高有效位。串行传输如图 1-5 所示。

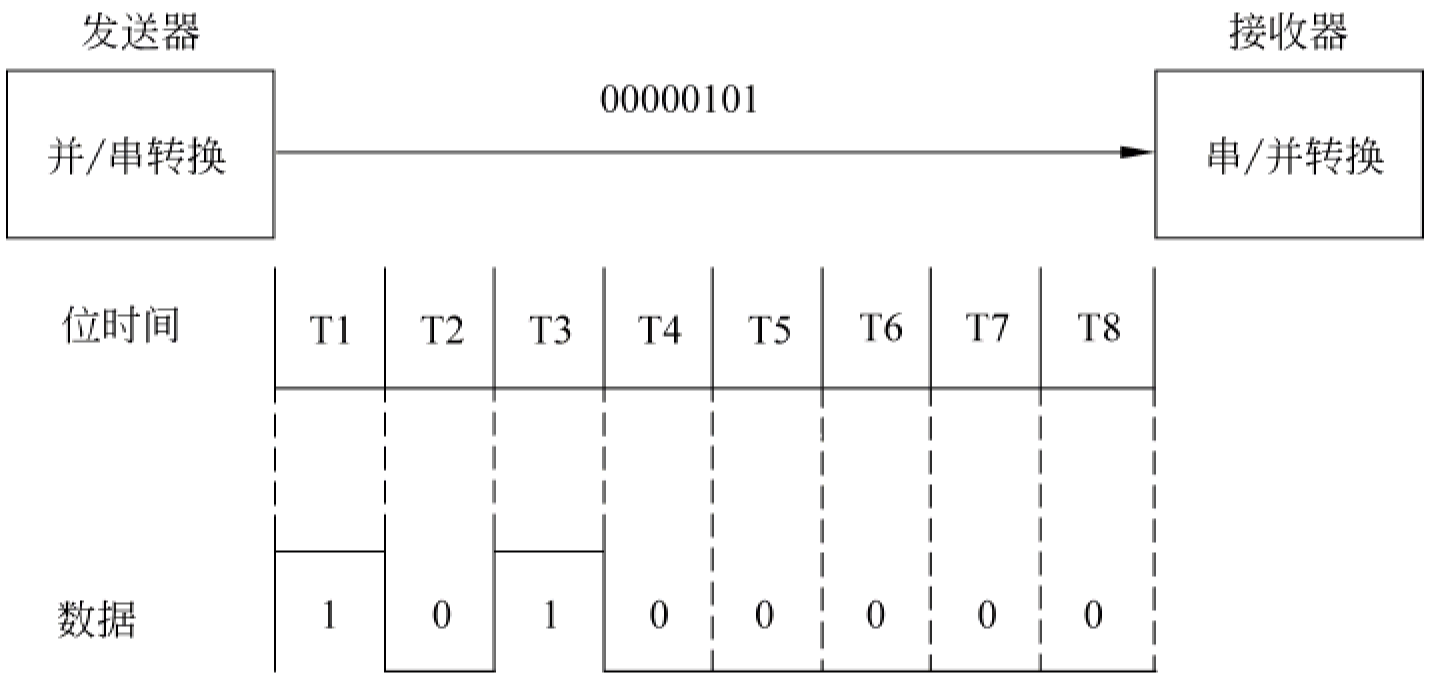


图 1-5 串行传输

当串行传输时,有可能按顺序连续传送若干个 0 或若干个 1。为了确定究竟传送了多少个连续的 0 或连续的 1,通常采用的方法是指定“位时间”,即指定一个二进制位在传输线上占用的时间长度,“位时间”通常用一个时钟信号来控制,时钟的频率决定了串行传输的速度。

在串行传输时,被传送的数据需要在发送部件进行并/串变换,而在接收部件又需要进行串/并变换。串行传输的主要优点是只需要一条传输线,这一点对长距离传输尤其重要,不管传送多少数据量都只需要一条传输线,成本比较低廉。

2) 并行传输

用并行方式传输二进制信息时,对应于每个数据位都需要一条单独的传输线,信息由多少二进制位组成,就需要多少条传输线,从而使得二进制数 0 或 1 在不同的线上同时进行传输,如图 1-6 所示。

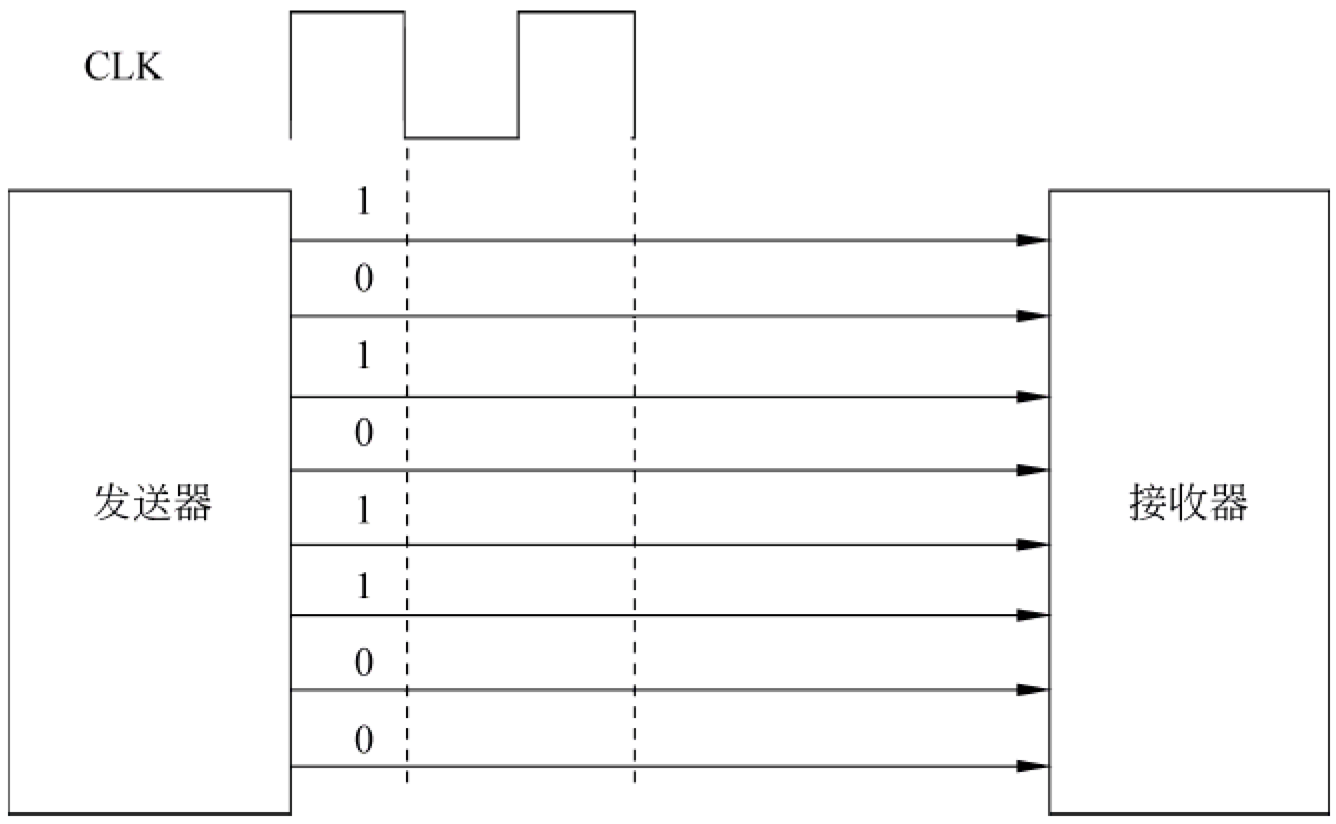


图 1-6　并行传输

很明显,并行通信的速度要比串行通信的速度要快,效率更高,费时更少。早期 I/O 为了提高效率多采用并行总线。但随着总线时钟的提高,在高速状态下,并行口的数据线之间存在串扰,且并行口需要信号同时发送同时接收,任何一根数据线的延迟都会引起问题。而串行只有一根数据线,还可以采用差分信号传输提高抗干扰性,所以可以实现更高的传输速率;因此尽管并行可以一次传多个数据位,但是时钟远远低于串行,所以目前串行传输是 CPU 和外设高速传输的首选方案。

1.2.2.4　总线的连接方式

总线的排列布置以及总线与其他各类部件的连接方式,对计算机系统性能有重要影响。根据连接方式的不同,微机系统中采用的总线结构可分成三种基本类型:单总线结构、双总线结构和三总线结构。

1) 单总线结构

在一些微机系统中,使用一条系统总线来连接 CPU、主存和 I/O 设备,称为单总线结

构,如图 1-7 所示。

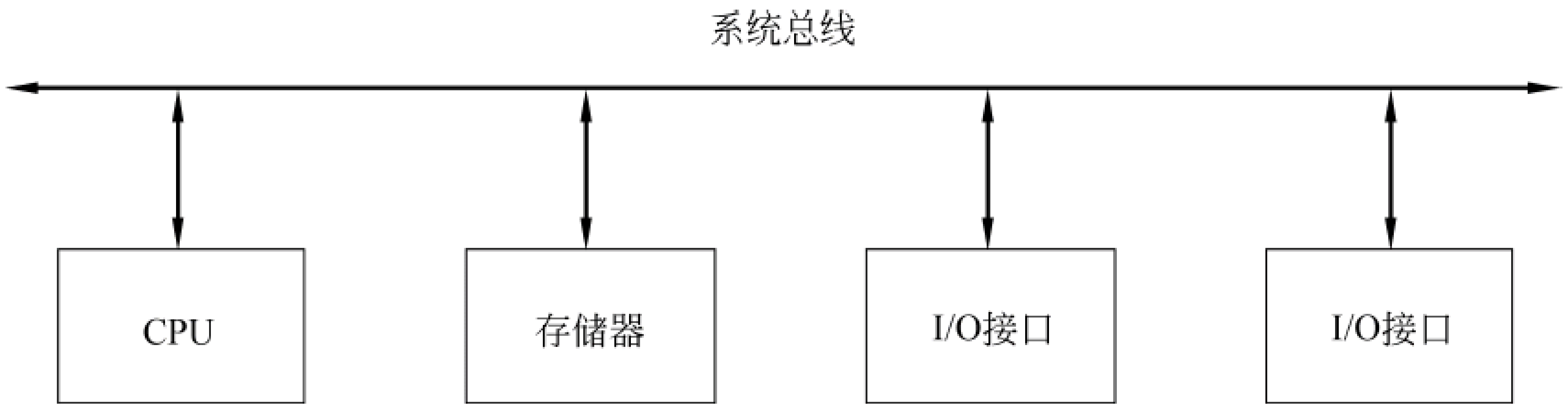


图 1-7　单总线结构

在单总线结构中,要求连接到总线上的逻辑部件都必须高速运行,以便在某些设备需要使用总线时能够迅速获得总线控制权,当不再使用总线时也能迅速放弃总线控制权。否则,由于一条总线由多个功能部件共用,有可能导致很大的时间延迟。

在单总线结构中,当 CPU 取一条指令时,首先把程序计数器(PC)中的地址同控制信息一起送至总线上。该地址不仅送至主存,同时也送至总线上的所有外围设备,只有与总线上的地址相对应的设备才执行数据传送操作。取指令情况下的地址是主存地址,因此该地址所指定的主存单元中的指令被传送给 CPU,CPU 检查指令中的操作码,确定对数据执行什么操作,以及数据是流进还是流出 CPU。

在单总线系统中,对输入输出设备的操作与主存的操作方法完全一样。当 CPU 把指令的地址字段送到总线上时,如果该地址字段对应的地址是外围设备地址,则外围设备予以响应,从而在 CPU 和对应的外围设备之间发生数据传送,数据传送的方向也由指令操作码决定。

单总线结构的优点在于容易扩展成多 CPU 系统,只要在系统总线上挂接多个 CPU 即可。但是,在单总线结构中,由于所有逻辑部件都挂在同一个总线上,因此总线只能分时工作,即某一个时间只能允许一对部件之间传送数据,这就使信息传送的吞吐量受到限制。

2) 双总线结构

图 1-8 为双总线结构,这种结构保持了单总线系统简单、易于扩充的优点,但又在 CPU 和主存之间专门设置了一组高速的存储总线,使 CPU 可通过专用的存储总线与存储器交换信息,以减轻系统总线的负担,同时主存仍可通过系统总线与外设进行 DMA(Direct Memory Access)操作,而不必经过 CPU。

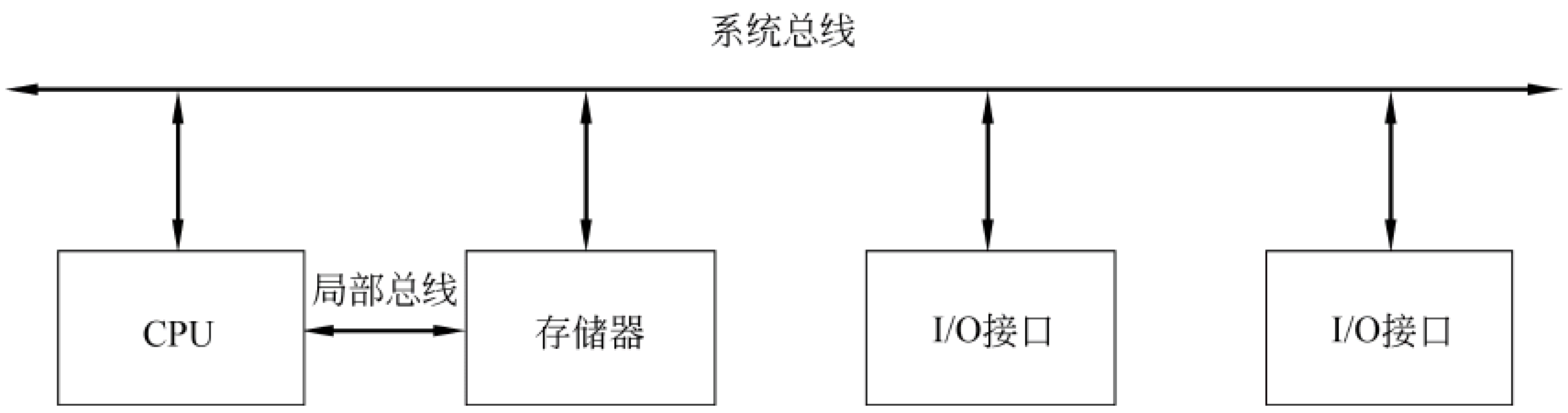


图 1-8　双总线结构

3）三总线结构

图 1-9 为三总线结构。三总线结构是在双总线系统的基础上增加 I/O 总线形成的，其中系统总线是 CPU、主存和 I/O 通道处理机（IOP）之间进行数据传送的公共通路，而 I/O 总线则是多个外围设备与通道之间进行数据传送的公共通路。

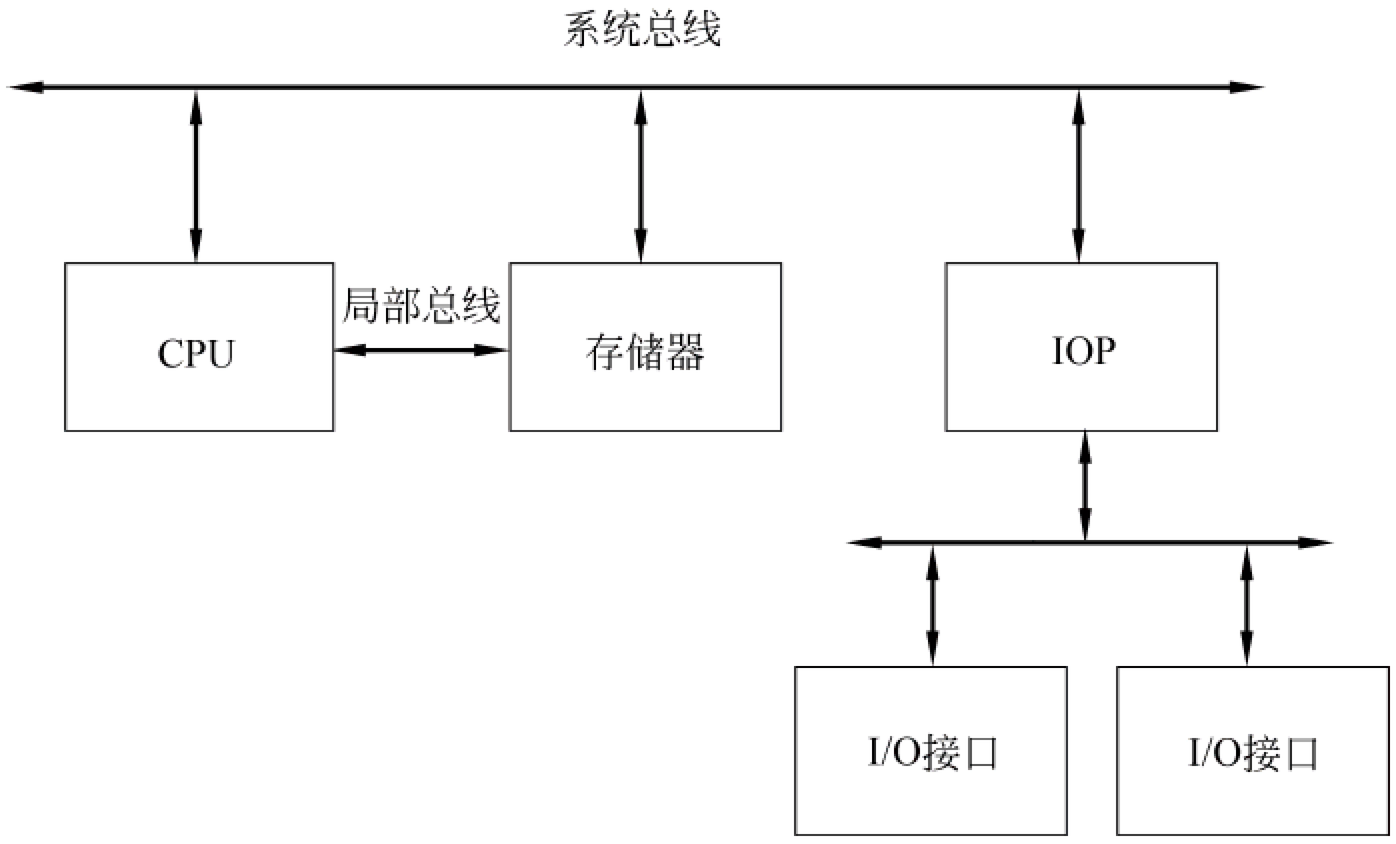


图 1-9　三总线结构

通道实际上是一台具有特殊功能的处理器，又称为 IOP（I/O Processor），它分担了 CPU 的一部分功能，实现对外设的统一管理，完成外设与主存之间的数据传送。这一思想与基于总线的网络将集线器（Hub）转换成交换机（Switch）以提高通信速率的思想是一致的。显然，由于增加了 IOP，整个系统的工作效率可以大大提高。

1.2.2.5　典型总线

1）工业标准结构总线（Industry Standard Architecture，ISA）

ISA 总线是由 IBM PC/XT 和 PC/AT 使用的 8 位总线发展而来的总线标准。ISA 是 8/16 位兼容总线，因此 I/O 插槽有 8 位和 8/16 位两种类型：8 位扩展槽由 62 个引脚组成，其中包括 20 条地址线和 8 条数据线，用于 8 位数据传输；8/16 位扩展槽除了一个 8 位 62 线的连接器外，还有一个附加的 36 线连接器，这种扩展插槽既可以支持 8 位插接板，也可支持 16 位插接板（24 条地址线和 16 条数据线）。ISA 总线的应用范围很广，一般用于连接中、低速 I/O 设备。

2）外部设备连接总线（Peripheral Component Interconnect，PCI）

PCI 总线是 1991 年由 Intel、IBM、Compaq、Apple 等几家公司联合推出的。PCI 是一个与处理器无关的高速外围总线，又是至关重要的层间总线，可支持 10 台外部设备，它采用同步时序协议和集中式仲裁策略，并具有自动配置能力。PCI 总线体系结构中有三种桥接器：HOST 桥、PCI-PCI 桥、PCI-LEGACY 桥。桥接器是一个总线连接和转换部件，可以把一条总线的地址空间映射到另一条总线的地址空间上，从而使系统中任意一个总线主设备都能看到同样的一份地址表。HOST 桥是 PCI 总线控制器和仲裁器。

3）PCI Express 总线

随着 Pentium 4 前端总线频率的迅速提高（高达 1GHz 以上），原有的 PCI 总线标准已难以适应新的要求。PCI Express 是一种基于串行技术、高带宽连接点、芯片到芯片连接的新型总线技术。有别于 PCI 并行技术，PCI Express 的一个通道采用 4 根信号线，两根差分信号线用于接收，另外两根差分信号线用于发送；信号频率 2.5GHz，采用 8/10 位编码；定义了用于多种通道的连接方式，如×1、×4、×8、×16 以及×32 通道的连接器，分别对应于 500MB/s、2GB/s、4GB/s、8GB/s 和 16GB/s 的带宽。采用 PCI Express 总线标准的最大意义在于其通用性和兼容性，通过与 PCI 软件模块的完全兼容，可以确保现有设备和驱动程序不用修改仍能正常工作。

4）通用串行总线（Universal Serial Bus，USB）

USB 总线是连接计算机系统与外部设备的一种串口总线标准，也是一种输入输出接口的技术规范，被广泛地应用于个人计算机和移动设备等信息通信产品，并扩展至摄影器材、数字电视（机顶盒）、游戏机等其他相关领域。

USB 最初是由英特尔公司与微软公司倡导发起，其最大的特点是支持热插拔和即插即用。当设备插入时，主机枚举到此设备并加载所需的驱动程序，因此在使用上远比 PCI 和 ISA 总线方便。USB 的设计为非对称式的，它由一个主机控制器和若干通过集线器设备以树形连接的设备组成。一个控制器下最多可以有 5 级 Hub，包括 Hub 在内，最多可以连接 128 个设备，一台计算机可以同时有多个控制器。USB 1.1 的最大传输带宽为 12Mb/s，USB 2.0 的最大传输带宽为 480Mb/s，USB 3.0 为 5Gb/s。最新一代是 USB 3.1，传输速度为 10Gb/s，三段式电压 5V/12V/20V，最大供电 100W，另外除了旧有的 Type-A、B 接口之外，新型 USB Type-C 接头不再分正反。

5）SATA 总线

SATA（Serial Advanced Technology Attachment）是一种替代并行 ATA 的总线技术。SATA 总线使用嵌入式时钟信号，具备了更强的纠错能力，与以往相比其最大的区别在于能对传输指令（不仅仅是数据）进行检查，如果发现错误会自动矫正，这在很大程度上提高了数据传输的可靠性。串行接口还具有结构简单、支持热插拔的优点。SATA 分别有 SATA 1.5Gb/s、SATA 3Gb/s 和 SATA 6Gb/s 三种规格，是目前 PC 硬盘连接的主流总线。

1.2.3　哈佛体系结构

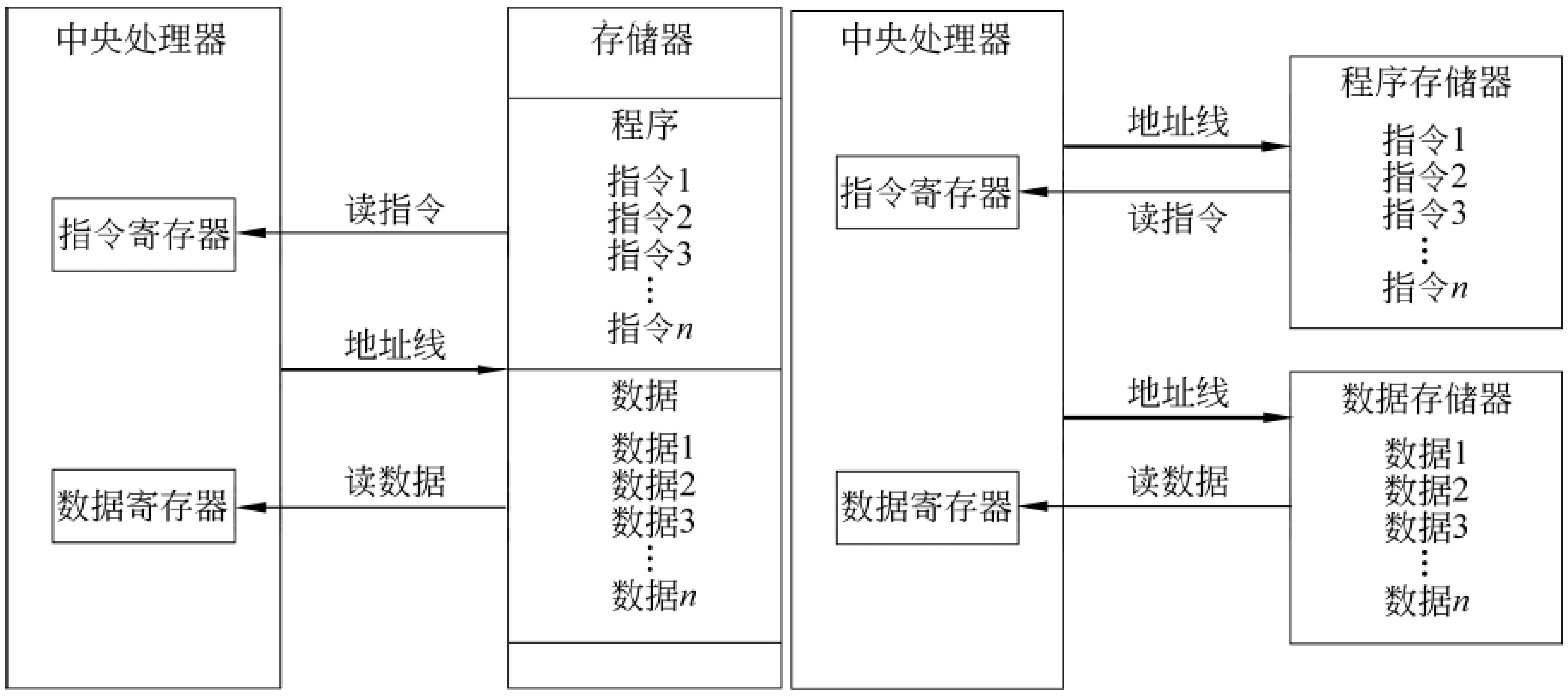
冯·诺依曼结构只有一个存储器，数据和指令存储在同一个存储器中，使用同一组地址线进行数据和指令的读写。代表性的处理器有 Intel 8086 及后续系列和 ARM7 等。

哈佛架构（Harvard Architecture）是一种将程序指令和数据分开的存储器结构。程序指令储存和数据储存具有独立的总线接口，数据和指令的访问可以同时进行。

哈佛架构的微处理器通常具有较高的执行效率，程序指令和数据指令分开组织和储存的，可以实现指令预取。目前，大多数处理器采用这种独立信号通路的结构。

如图 1-10 所示，纯冯·诺依曼架构下的 CPU 可以读取指令或读写主存数据，指令和数

据分时共享相同的总线。使用哈佛结构的 CPU，即使没有缓存的情况下，也可以读取指令的同时进行数据访问，因此，哈佛结构的计算机可以在相同的电路复杂度下有更好的性能表现，哈佛架构拥有不同的代码和数据的地址空间：即指令的零地址和数据的零地址是不同的。



（a）冯 · 诺依曼结构　　（b）哈佛结构

图 1-10　冯 · 诺依曼结构与哈佛结构的区别

现代高性能 CPU 芯片在设计上融合了哈佛结构和冯 · 诺依曼结构的特点，一种典型的方式是将 CPU 的缓存分为指令缓存和数据缓存两部分，CPU 访问缓存时使用哈佛体系结构；然而当缓存未命中时，数据从主存储器中读取，此时并不分为独立的指令和数据部分。

哈佛结构主要用于数字信号处理器（DSP）和微控制器。DSP 一般执行高度优化的音频或视频处理算法，通常采用单指令多数据流（SIMD）和超长指令字（VLIW）等架构，一般具有多套总线实现数据的并行读写，例如 TI 的 TMS320 C55x 处理器，具有多个并行数据总线（双写、三读）和指令总线。

微控制器的特点是具有少量的程序存储器（闪存）和数据存储器（SRAM），较少有缓存，大多利用哈佛架构的并行高速处理指令和数据的访问。分开存储的程序和数据存储器可能具有不同的位宽，例如使用 16 位指令和 8 位宽的数据，如 Atmel 的 AVR 和 Microchip 的 PIC 系列。

1.2.4　微处理器的内部结构

1.2.4.1　基本结构

传统上，CPU 由控制器和运算器这两个主要部件组成。随着集成电路技术的不断发展和进步，新型 CPU 纷纷集成了一些原先置于 CPU 之外的分立功能部件，如浮点处理器、高速缓存等，在大大提高 CPU 性能指标的同时，也使得 CPU 的内部组成日益复杂化。

1. 控制器

控制器是整个计算机系统的指挥中心。在控制器的指挥控制下，运算器、存储器和输入

输出设备等部件协同工作。

控制器根据程序预定的指令执行顺序，从主存取出一条指令，按照该指令的功能，控制计算机内各功能部件的操作，协调和指挥整个计算机实现指令的功能。

控制器通常由程序计数器（PC）、指令寄存器（IR）、指令译码器（ID）和操作控制器组成。其主要功能包括：

（1）从主存中取出一条指令，并指出下一条指令在主存中的位置；

（2）对指令进行译码，并产生相应的操作控制信号，以便启动规定的动作；

（3）指挥并控制 CPU、主存和输入输出设备之间数据流动的方向。

2. 运算器

运算器是计算机中用于实现数据加工处理功能的部件，它接受控制器的命令，负责完成对操作数据的加工处理任务，其核心部件是算术逻辑单元 ALU。运算器接受控制器的命令而进行动作，即运算器所进行的全部操作都是由控制器发出的控制信号来指挥的，所以它是执行部件。

运算器由算术逻辑单元（ALU）、寄存器、程序状态寄存器等组成。它有两个主要功能：

（1）执行所有的算术运算；

（2）执行所有的逻辑运算，并进行逻辑测试。

3. 寄存器

在 CPU 中至少要有五类寄存器：指令寄存器（IR）、程序计数器（PC）、地址寄存器（AR）、数据寄存器（DR）、程序状态寄存器（PSR）。这些寄存器用来暂存一个计算机的字，其数目可以根据需要进行扩充。

1）数据寄存器（Data Register，DR）

数据寄存器又称数据缓冲寄存器，其主要功能是作为 CPU 和主存、外设之间信息传输的中转站，用于弥补 CPU 和主存、外设之间操作速度上的差异。

数据寄存器用来暂时存放由主存储器读出的一条指令或一个数据字；反之，当向主存存入一条指令或一个数据字时，也将它们暂时存放在数据寄存器中。

数据寄存器的作用：

- 作为 CPU 和主存、外围设备之间信息传送的中转站；
- 弥补 CPU 和主存、外围设备之间在操作速度上的差异。

2）指令寄存器（Instruction Register，IR）

指令寄存器用来保存当前正在执行的一条指令。当执行一条指令时，首先把该指令从主存读取到数据寄存器中，然后再传送至指令寄存器。

指令包括操作码和操作数/地址码两个字段，为了执行指令，必须对操作码进行测试，识别出所要求的操作，指令译码器（Instruction Decoder，ID）用于完成这项工作。指令译码器对指令寄存器的操作码部分进行译码，以产生指令所要求操作的控制电位，在时序部件定时信号的作用下，产生具体的操作控制信号。

3）程序计数器（Program Counter，PC）

程序计数器用来指出下一条指令在主存储器中的地址。在程序执行之前，首先必须将

程序的首地址，即程序第一条指令所在主存单元的地址送入 PC。当执行指令时，CPU 能自动递增 PC 的内容，使其始终保存将要执行的下一条指令的主存地址，为取下一条指令做好准备。若为单字节指令，则 PC+1；若为双字节指令，则 PC+2，以此类推。

但是，当遇到转移指令时，下一条指令的地址将由转移指令的地址码字段来指定，即 PC 寄存器的值由指令所带的地址决定而非通过顺序递增 PC 的内容来取得。

4）地址寄存器（Address Register，AR）

地址寄存器用来保存 CPU 当前所访问的主存单元的地址。由于在主存和 CPU 之间存在操作速度上的差异，所以必须使用地址寄存器来暂时保存主存的地址信息，直到主存的存取操作完成为止。当 CPU 和主存进行信息交换，即 CPU 向主存存入数据/指令或者从主存读出数据/指令时，都要使用地址寄存器和数据寄存器。如果我们把外围设备与主存单元进行统一编址，那么，当 CPU 和外围设备交换信息时，我们同样要使用地址寄存器和数据寄存器。

5）程序状态寄存器（Program Status Register，PSR）

程序状态寄存器用来保存由算术/逻辑指令运行或测试的结果所建立起来的各种条件码内容，如运算结果进/借位标志（C）、运算结果溢出标志（O）、运算结果为零标志（Z）、运算结果为负标志（N）、运算结果符号标志（S）等。除此之外，程序状态寄存器还用来保存中断和系统工作状态等信息，以便 CPU 和系统及时了解机器运行状态和程序运行状态。

1.2.4.2 流水线

早期的计算机采用的是串行处理，计算机的各个操作只能串行地完成，即任一时刻只能进行一个操作。并行处理使得多个操作能同时进行，从而大大提高了计算机的速度。并行处理的主要方法包括如下。

（1）时间并行技术：时间并行指分时，即让多个处理过程在时间上相互错开，轮流重叠地使用同一套硬件设备的各个部分，以加快硬件周转而赢得速度。分时并行的实现方式就是流水线，目前的高性能计算机几乎无一例外地使用了流水线技术。

（2）空间并行技术：空间并行指硬件资源重复，即以多个相同的硬件来大幅度提高计算机的处理速度。空间并行技术主要体现在多核处理器、多处理器系统，超算就是典型的空间并行技术。

（3）时间空间并行技术：即综合使用了分时和硬件并行，高性能微处理器一般都采用时间空间并行技术。

计算机的流水线（Pipeline）工作方式就是将一个计算任务细分成若干子任务，每个子任务都由专门的功能部件进行处理，一个计算任务的各个子任务由流水线上各个功能部件轮流进行处理，最终完成工作。这样，不必等到上一个计算任务完成，就可以开始下一个计算任务的执行。

流水线的硬件基本结构如图 1-11 所示。流水线由一系列串联的功能部件（Si）组成，各个功能部件之间设有高速缓冲寄存器（L），以暂时保存上一功能部件对子任务处理的结果，同时又能够接受新的处理任务。在一个统一的时钟（C）控制下，计算任务从功能部件的一

个功能段流向下一个功能段。在流水线中，所有功能段同时对不同的数据进行不同的处理，各个处理步骤并行地操作。

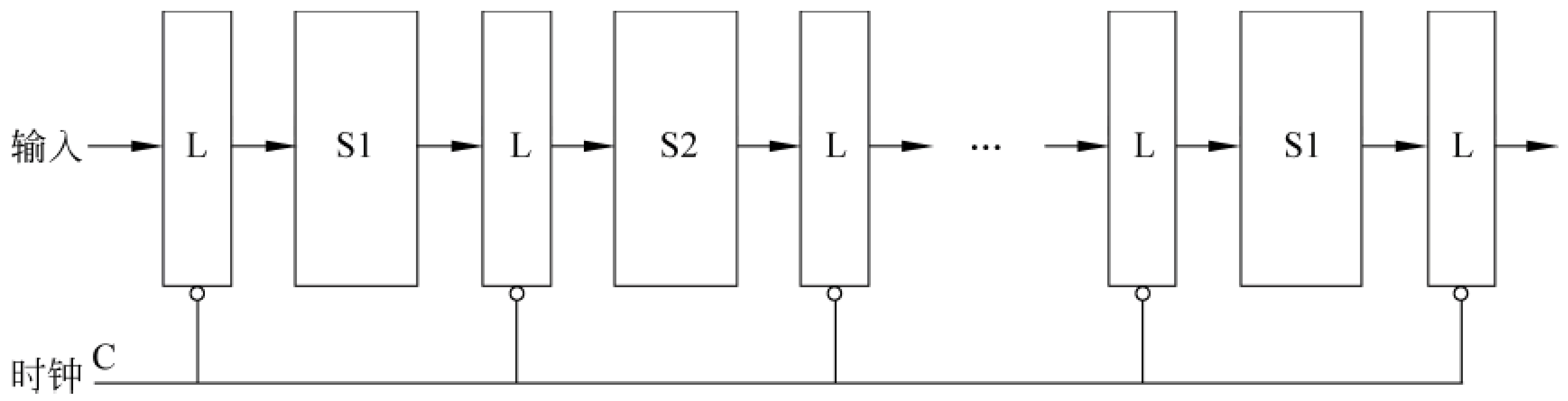


图 1-11　流水线的硬件基本结构

理想情况下，每步需要一个时钟周期。当流水线完全装满时，每个时钟周期平均有一条指令从流水线上执行完毕，输出结果。当指令流不能按流水顺序执行时，流水过程会中断（即断流）。在一个流水过程中，流水线的性能取决于流水部件中最慢的部件，因此实现各个子过程的各个功能段所需要的时间应该尽可能保持相等，以避免产生瓶颈。

假设一个指令的执行周期包含取指（IF）、译码（ID）、执行（EX）、访存（MEM）、写回（WB）5 个过程，最慢的部件执行时间记为一个单位时间，在不采用流水线时的执行过程如图 1-12 所示。上一条指令的 5 个子过程全部执行完毕后才能开始下一条指令，每隔 5 个单位时间才有一个输出结果，15 个单位时间完成 3 条指令，每条指令平均用时 5 个单位时间。

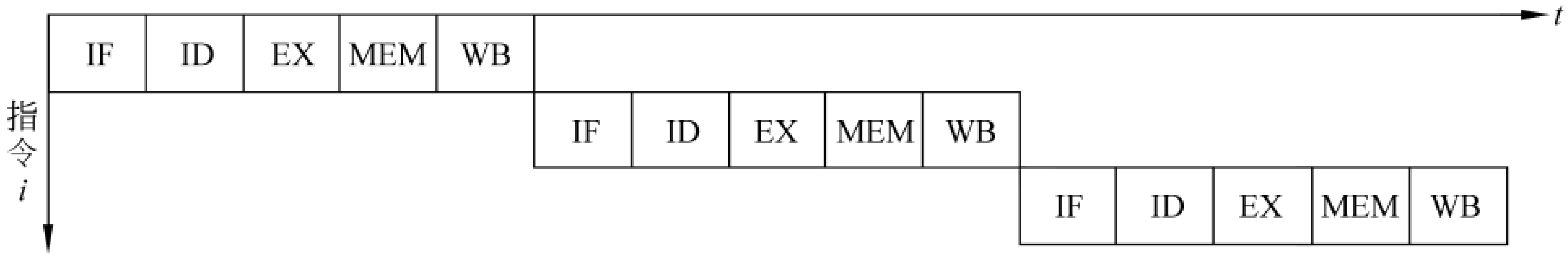


图 1-12　非流水顺序执行过程

采用 5 级流水线后指令的执行过程如图 1-13 所示，上一条指令与下一条指令的 5 个子过程在时间上可以重叠执行，当流水线满载时，每一个单位时间就可以输出一个结果。因此，9 个单位时间完成了 5 条指令，每条指令平均用时 1.8 个单位时间。虽然每条指令的执行时间并未缩短，但 CPU 运行指令的总体速度却能成倍提高。

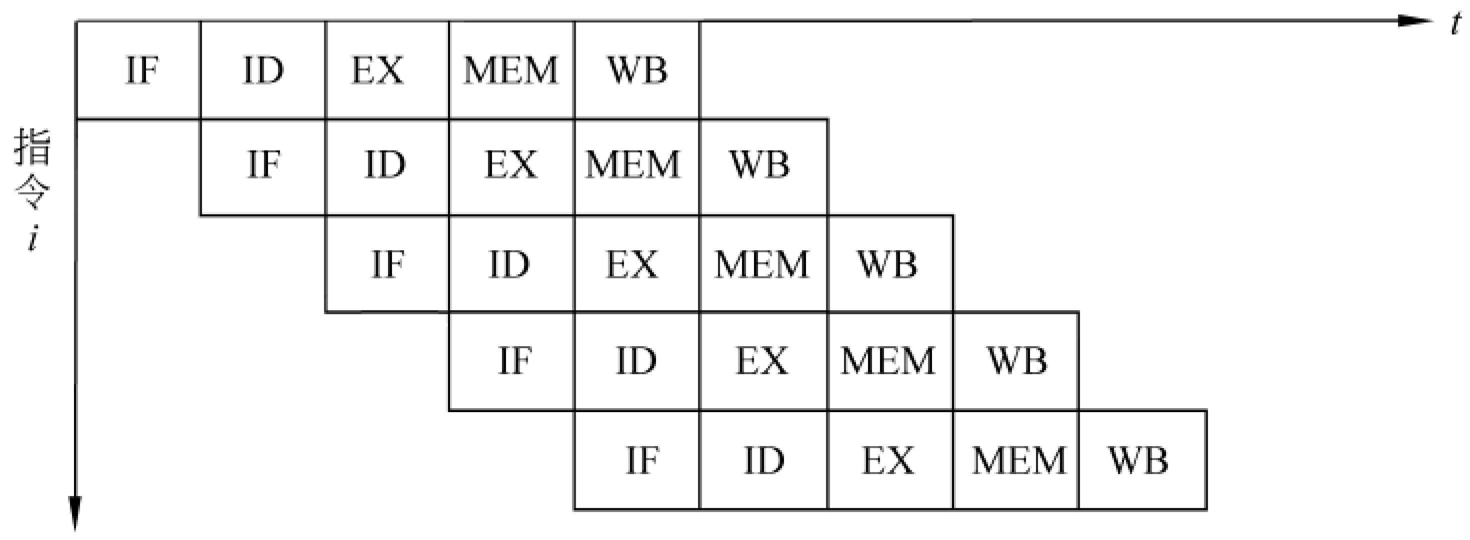


图 1-13　5 级流水执行过程

流水线的每个阶段的计算结果在周期结束以前都要发送到阶段之间的缓冲器上，以供下一个阶段使用。所以，单位时间就是由以上几个阶段中的耗时最长的那个决定的。假设耗时最长的阶段耗时为 s 秒，那么时钟频率最多就只能设计到 1s/Hz。要提高时钟频率，一种最简单的办法，就是将每个阶段再进行细分成更小的步骤，细分后的每个阶段，单个阶段的运算量小了，单位耗时 s 也就减少，这样实际上就是提高了时钟频率。这种将标准流水线细分的技术，就是超级流水线技术。比如，Pentium M 有 14 级流水线，Pentium 4 有 31 级流水线。

只有一条指令流水线的计算机称为标量流水计算机，如果计算机具有两条以上的流水线，称为超标量计算机，图 1-14 表示超标量流水计算机的时空图。当流水线满载时，每一个时钟周期可以执行 2 条以上的指令。超标量流水计算机是时间并行技术和空间并行技术的综合应用。

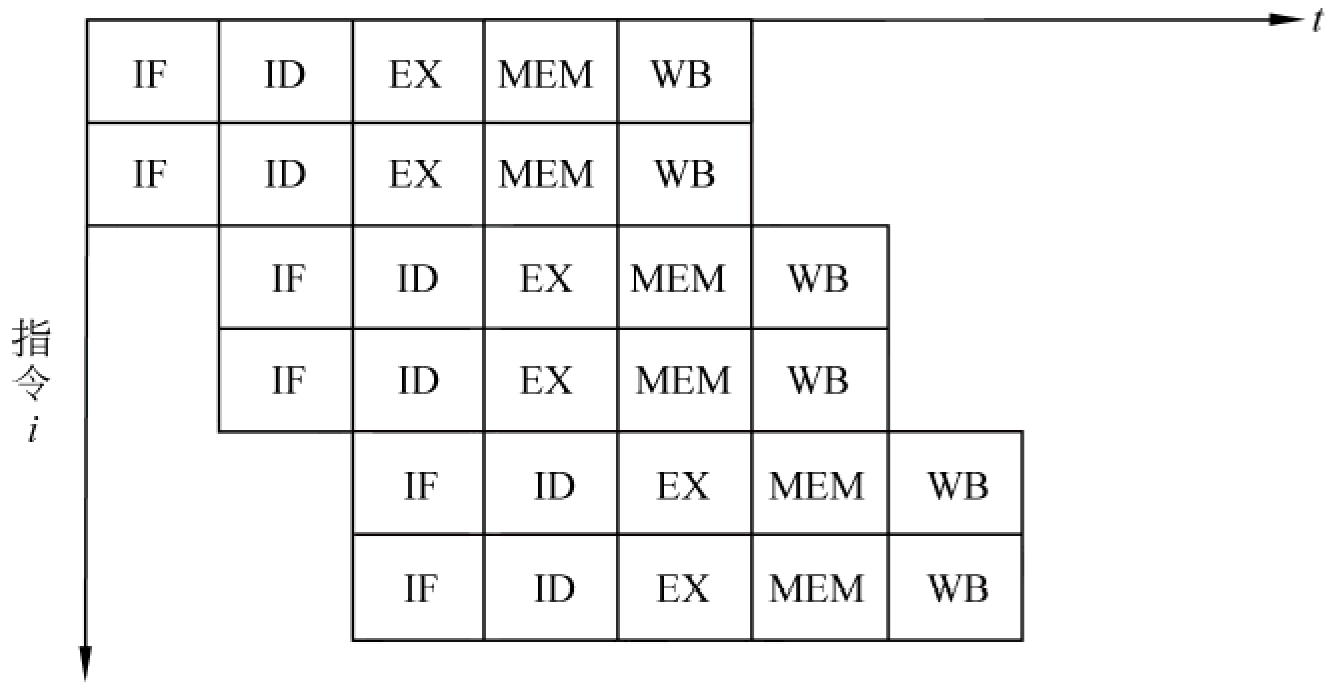


图 1-14　超标量流水线执行

要使流水线发挥高效率，就要使流水线连续不断地流动，尽量不出现断流。然而，流水线中的各条指令之间存在一些相关性，即第二条指令的执行必须要使用第一条指令的执行结果，使得指令的执行受到影响，断流不可避免；此外，跳转指令也会引起流水线断流，现代计算机通常采用乱序执行和分支预测等进行执行优化，以降低断流引起的性能损失。

【**思考题**：硬件多线程是什么？属于空间并行还是时间并行?】

1.2.5　I/O 接口技术

一般而言，CPU 管理外围设备的输入输出控制方式有 5 种：查询方式、中断方式、DMA 方式、通道方式、外围处理机方式。第一种方式由软件实现，后四种方式需要特殊硬件的支持才能实现。目前微机系统中常用的是查询方式、中断方式和 DMA 方式。

1）查询方式

程序查询方式是早期计算机中使用的一种方式，CPU 与外围设备的数据交换完全依赖于计算机的程序控制。在进行信息交换之前，CPU 要设置传输参数、传输长度等，然后启动外设工作，与此同时，外设则进行数据传输的准备工作；相对于 CPU 来说，外设的速度是比

较低的，因此外设准备数据的时间往往是一个漫长的过程，而 CPU 不知道数据何时准备好，在这段时间里，CPU 除了循环检测外设是否已准备好之外，不能处理其他业务，只能一直等待，直到外设完成数据准备工作，CPU 才能开始进行信息交换。

查询方式的优点是 CPU 的操作和外围设备的操作能够完全同步，不需要额外硬件。但是，由于外围设备的动作通常很慢，程序进行循环查询浪费 CPU 时间，数据传输效率低下，CPU 利用率很低。在当前的实际应用中，除了简单的嵌入式系统应用之外，已经很少使用程序查询方式了。

【思考题：从用户和操作系统的角度分别考虑，CPU 利用率高好还是低好？】

2）中断方式

中断是外围设备用来主动通知 CPU，准备发送或接收数据的一种方式。当一个中断发生时，CPU 暂停其现行程序，转而执行中断处理程序，完成数据 I/O 工作；当中断处理完毕后，CPU 又返回到原来的任务，并从暂停处继续执行程序。这种方式节省了 CPU 时间，实现了外设和 CPU 的并行工作，是管理 I/O 操作的一个比较有效的方法，现代计算机中，中断是必不可少的 I/O 机制，通常有中断控制器对中断进行管理。中断方式一般适用于随机出现的服务请求，并且对响应速度有一定的要求。

3）直接存储器存取方式 DMA

直接存储器存取方式（DMA）是一种完全由硬件执行 I/O 交换的工作方式。DMA 控制器从 CPU 完全接管对总线的控制权，数据交换不经过 CPU 而直接在主存和外围设备之间进行，以便高速传送数据。这种方式的主要优点是数据传送效率很高，传送速率仅受限于主存的访问时间，CPU 只需要配置 DMA 数据传输的起始地址，目的地址和数量，数据的传输完全由 DMA 控制总线完成，并通过中断方式通知 CPU。与程序中断方式相比，这种方式需要特殊的硬件支持，适用于主存和高速外围设备之间大批量数据交换的场合。

4）通道方式

DMA 方式的出现减轻了 CPU 对 I/O 操作的控制，使得 CPU 的效率显著提高，而通道的出现则进一步提高了 CPU 的效率。通道分担了 CPU 的一部分功能，可以实现对外围设备的统一管理，完成外围设备与主存之间的数据传送。

5）外围处理机方式

外围处理机（Peripheral Processor Unit，PPU）方式是通道方式的进一步发展。PPU 基本上独立于主机工作，它的结构更接近于一般的处理机，甚至就是微小型计算机。在一些系统中，设置了多台 PPU，分别承担 I/O 控制、通信、维护诊断等任务，从某种意义上说，这种系统已经变成了分布式多机系统。

1.2.6 存储器

冯·诺依曼计算机以存储器为中心，必须先把有关程序和数据装到存储器中，程序才能开始运行。在程序执行过程中，CPU 所需的指令、运算器所需的数据要从存储器中取出，运算结果必须在程序执行完毕之前全部写到存储器中，各种输入输出设备也直接与存储器交

换数据。因此,在计算机运行过程中,存储器是各种信息存储和交换的中心。

1. 存储器指标

存储器用于存储二进制数据,其最小的存储单位是一比特,基于地址总线宽度和存储代价的考虑,计算机系统中以 8 个二进制位即一字节(Byte)作为存储的基本单元,存储器的容量以字节为单位。对于大容量的存储,常用 KB、MB、GB、TB 来表示。

【**思考题**:以 Byte 作为基本存储单位合理吗?】

CPU 向存储器读写数据时,通常以字(Word)为单位,字由若干个字节组成,一个字到底等于多少个字节取决于计算机的字长,对于 32 位机,1 Word=4Byte=32bit。

衡量存储器的指标主要包括以下几点。

1) 存储容量

在一个存储器中可以容纳的存储单元的总数称为存储容量。在按字节寻址的计算机中,存储容量的最大字节数可由地址码的位数来确定。例如,一台计算机的地址码为 n 位,则可产生 2^n 个不同的地址码,则其最大容量为 2^n 字节。一台计算机设计定型以后,其地址总线、地址译码范围也已确定,因此其最大存储容量是确定的,通常情况下主存储器的实际存储容量小于理论上的最大容量。

【**思考题**:对于辅存,如硬盘,存储容量和地址线有何关系?】

2) 存取时间

存取时间又称为存储器访问时间或读写时间,是指从启动一次存储器操作到完成该操作所经历的时间。从一次读操作命令发出到该操作完成,将数据读入数据缓冲寄存器为止所经历的时间,为存储器读取时间。存储器从接受写命令到把数据从存储器数据寄存器的输出端传送到存储单元所需的时间,即为存储器的写入时间。

3) 存储周期

存储周期又称为访问周期,是指连续启动两次独立的存储器操作所需间隔的最小时间,它是衡量主存储器工作性能的重要指标。存储周期通常略大于存取时间。

【**思考题**:参考 DRAM 的特点,为何存储周期通常大于存取时间?】

4) 存储器带宽

存储器带宽是指单位时间里存储器所存取的信息量,是衡量数据传输速率的重要指标,通常以位/秒(b/s)或字节/秒(B/s)为单位,与存储器接口的数据总线宽度和存储周期有关。例如,总线宽度为 32 位,存储周期为 250ns,则存储器带宽=32b/250ns=128Mb/s。

2. 存储器分类

根据存储介质,存储器分为半导体存储器、磁盘存储器和光盘存储器等。按照存取方式可分为随机存储和顺序存储。按存储器的读写功能可分为只读存储器(Read Only Memory,ROM)和随机读写存储器(Random Access Memory,RAM)。

1) 随机读写存储器

存储单元的内容可按需随意取出或存入,且存取的速度与存储单元的位置无关的存储器。这种存储器在断电时将丢失其存储内容,故主要用于存储短时间使用的程序。按保存数据的机理,随机存储器又分为静态随机存储器(Static RAM,SRAM)和动态随机存储器

(Dynamic RAM,DRAM)。

静态随机存储器(SRAM):只要不断电信息就不会丢失。静态存储器的集成度低,成本高,功耗较大,通常作为 Cache 的存储体。

动态随机存储器(DRAM):采用 MOS 管和电容存储电荷来保存信息,使用时需要不断给电容充电才能保持信息。动态存储器电路简单,集成度高,成本低,功耗小,但需要反复进行刷新操作,工作速度较慢,适合作为主存储器的主体部分。

2) 只读存储器

只读存储器 ROM 是一种存储固定信息的存储器,其特点是在正常工作状态下只能读取数据,不能随时修改或重新写入数据。只读存储器电路结构简单,且存放的数据在断电后不会丢失,特别适合于存储永久性的、不变的程序代码或数据。只读存储器包括不可重写只读存储器和可重写只读存储器(PROM)两大类。我们目前在微机系统里常用的可重写只读存储器主要包括电擦除可编程 ROM(EEPROM)和闪存(Flash ROM)两种。

Flash ROM 中的内容或数据不像 RAM 一样需要电源支持才能保存,但又像 RAM 一样具有可重写性:在某种低电压下,其内部信息可读不可写,类似于 ROM,而在较高的电压下,其内部信息可以更改和删除,类似于 RAM。由于单片 Flash 存储容量大,易于修改,Flash ROM 也常用于数码相机、U 盘以及固态硬盘中。

按存储器在计算机系统中的作用分,存储器可分为主存储器和辅助存储器。CPU 能直接访问的存储器称为内存储器(简称内存),或主存储器。CPU 不能直接访问的存储器称为外存储器(简称外存)或辅助存储器,外存的信息必须调入内存才能被 CPU 使用。

高速缓冲存储器(Cache)是计算机系统中的一个高速、小容量的半导体存储器,通常采用 SRAM,它位于高速的 CPU 和低速的主存之间,用于匹配两者的速度,达到高速存取指令和数据的目的。和主存相比,Cache 的存取速度快,但存储容量小。

内存储器用来存放计算机正在执行的大量程序和数据。

外存储器用于存放系统中的程序、数据文件及数据库。与内存相比,外存的特点是存储容量大,位成本低,但访问速度慢。

3. 分级存储体系

计算机对存储器的要求是容量大、速度快、成本低,需要尽可能地同时兼顾这三方面的要求。但是一般来讲,存储器速度越快,价格也越高,因而也越难满足大容量的要求。目前通常采用多级存储器体系结构,使用高速缓冲存储器、主存储器和外存储器,如图 1-15 所示。

由 Cache 和主存储器构成的 Cache-主存系统,其主要目标是利用与 CPU 速度接近的 Cache 来高速存取指令和数据以提高存储器的整体速度,从 CPU 角度看,Cache-主存的存储体系的访存速度接近 Cache,而容量是主存的容量。由主存和外存构成的虚拟存储器系统,其主要目的是增加存储系统的容量,从整体上看,其速度接近于主存的速度,其容量则接近于外存的容量。计算机存储系统的这种多层次结构,很好地解决了容量、速度、成本三者之间的矛盾。这些不同速度、不同容量、不同价格的存储器,用硬件、软件或软硬件结合的方式连接起来,形成一个系统。这个存储系统对应用程序员而言是透明的,在应用程序员看来

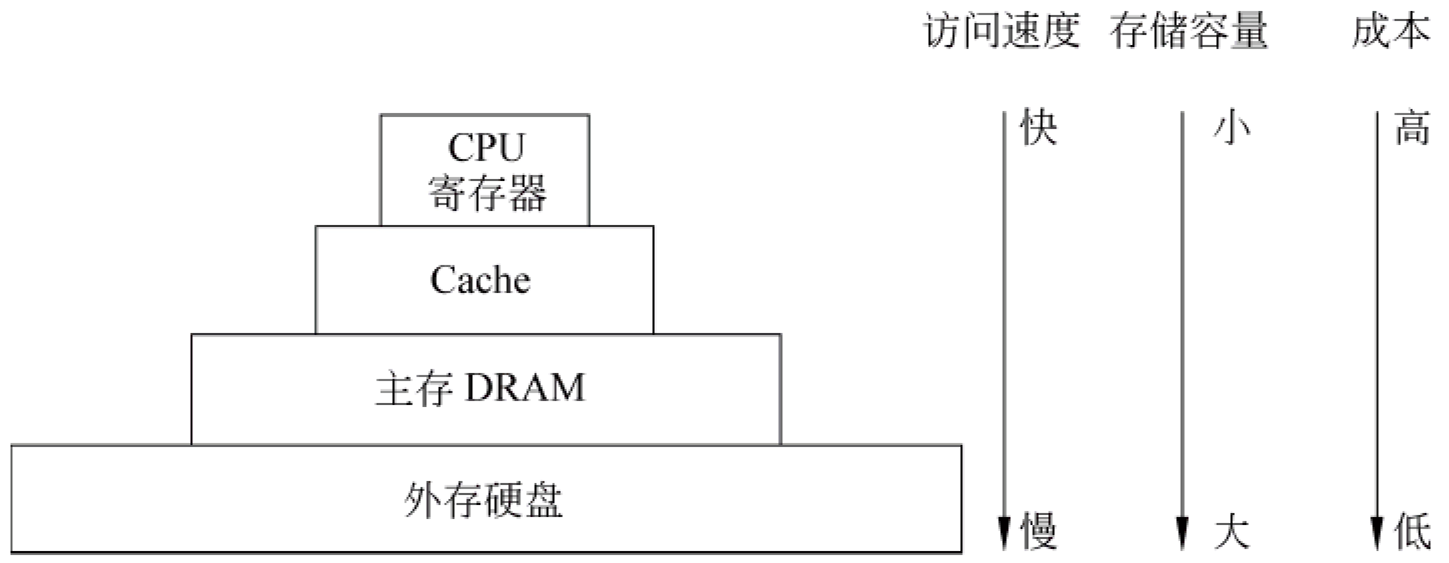


图 1-15　分层存储体系

它是一个存储器，其速度接近于最快的那个存储器，存储容量接近于容量最大的那个存储器，单位价格则接近最便宜的那个存储器。

1）Cache 缓存技术

设计和开发系统程序和应用程序时，程序员通常采用模块化的程序设计方法。某一模块的程序，往往集中在存储器逻辑地址空间中很小的一块范围内，且程序地址分布是连续的。也就是说，CPU 在一段较短的时间内，是对连续地址的一段很小的主存空间频繁地进行访问，而对此范围以外地址的访问较少，这种现象称为程序访问的局部性。

Cache 技术就是利用程序访问的局部性原理，把程序中正在使用的部分（活跃块）存放在一个小容量的高速 Cache 中，使 CPU 的访存操作大多针对 Cache 进行，从而解决高速 CPU 和低速主存之间速度不匹配的问题，使程序的执行速度大大提高。

Cache 是主存的缓冲存储器，由高速的 SRAM 组成，所有控制逻辑全部由硬件实现，对程序员而言是透明的。随着半导体器件集成度的不断提高，当前有些 CPU 已内置 Cache，并且出现了两级以上的多级 Cache 系统。

CPU 与 Cache 之间的数据交换是以字为单位的，而 Cache 与主存之间的数据交换则是以块为单位的。一个块由若干字组成。当 CPU 读取主存中的一个字时，该字的主存地址被发给 Cache 和主存，此时，Cache 控制逻辑依据地址判断该字当前是否存在于 Cache 中：若在，该字立即被从 Cache 传送给 CPU；若不在，则用主存读周期把该字从主存读出送到 CPU，同时把含有这个字的整个数据块从主存读出送到 Cache 中，并采用一定的替换策略将 Cache 中的某一块替换掉，替换算法由 Cache 管理逻辑电路来实现。

2）虚存技术

由于工艺和成本的原因，主存的容量受到限制。然而，计算机系统软件和应用软件的功能不断增强，程序规模迅速扩大，要求主存的容量越大越好，为此，操作系统将部分外存作为主存使用，即一个程序的部分地址空间在主存，另一部分在外存，当所访问的信息不在主存时，则由操作系统来安排 I/O 指令，把信息从外存调入主存。从效果上来看，好像为用户提供了一个存储容量比实际主存大得多的存储器，用户无需考虑所编程序在主存中是否放得下或放在什么位置等问题，这种存储器称为虚拟存储器。虚拟存储器只是一个容量非常大的存储器的逻辑模型，不是任何实际的物理存储器。虚拟存储技术中程序指令采用虚拟地址（或者叫逻辑地址），程序运行时，CPU 以虚拟地址来访问主存，由

辅助硬件找出虚拟地址和实际物理地址之间的对应关系，并判断这个虚拟地址指示的存储单元内容是否已装入主存。如果已在主存中，则通过地址变换，CPU 可直接访问主存的实际单元；如果不在主存中，则把包含这个字的一个外存存储块调入主存后再由 CPU 访问。如果主存已满，则由替换算法从主存中将暂不运行的一块调回外存，再从外存调入新的一块到主存。

1.2.7 程序的执行过程

CPU 的基本工作是执行预先存储的指令序列。程序的执行过程实际上是不断地取出指令、分析指令、执行指令的过程。

CPU 从存放程序的主存储器里取出一条指令，译码并执行这条指令，保存执行结果，紧接着又去取指令，译码，执行指令……如此周而复始，反复循环，使得计算机能够自动地工作，直到遇到停止指令，其过程如图 1-16 所示。

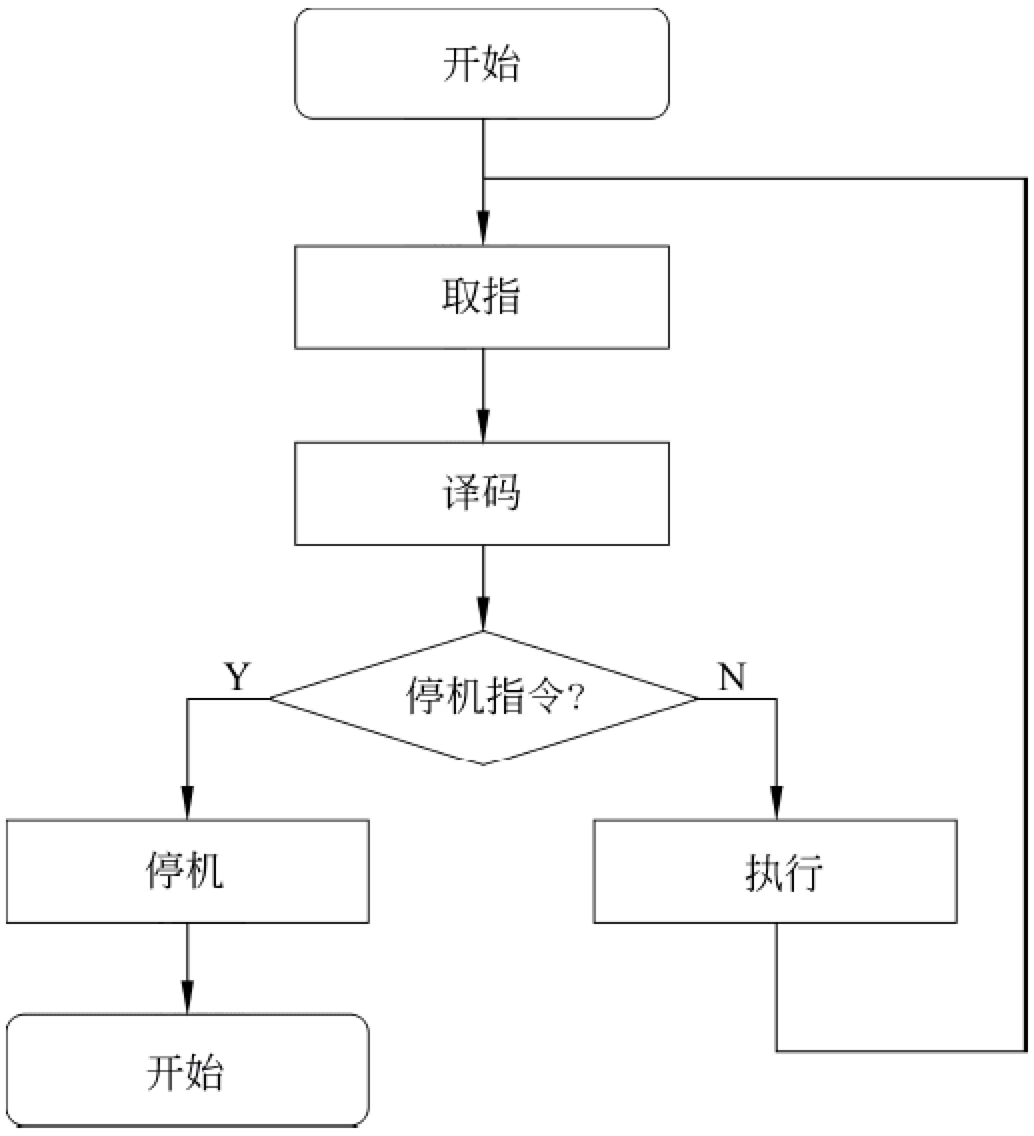


图 1-16 程序执行流程

CPU 是在时钟的驱动下工作的，时钟周期是处理操作的最基本时间单位，由机器的主频决定。机器内部各种操作大致可归属为 CPU 内部操作和对主存的操作两大类。从内存读取一条指令字的最短时间定义为 CPU 周期(也叫机器周期)，由于 CPU 内部操作速度较快，CPU 访问一次内存所花的时间较长，CPU 周期包含若干个时钟周期。CPU 取出一条指令并执行该指令所需的时间，称为指令周期，指令周期的长短与指令的复杂程度有关，一般是若干个 CPU 周期。时钟周期、机器周期和指令周期之间的关系如图 1-17 所示。

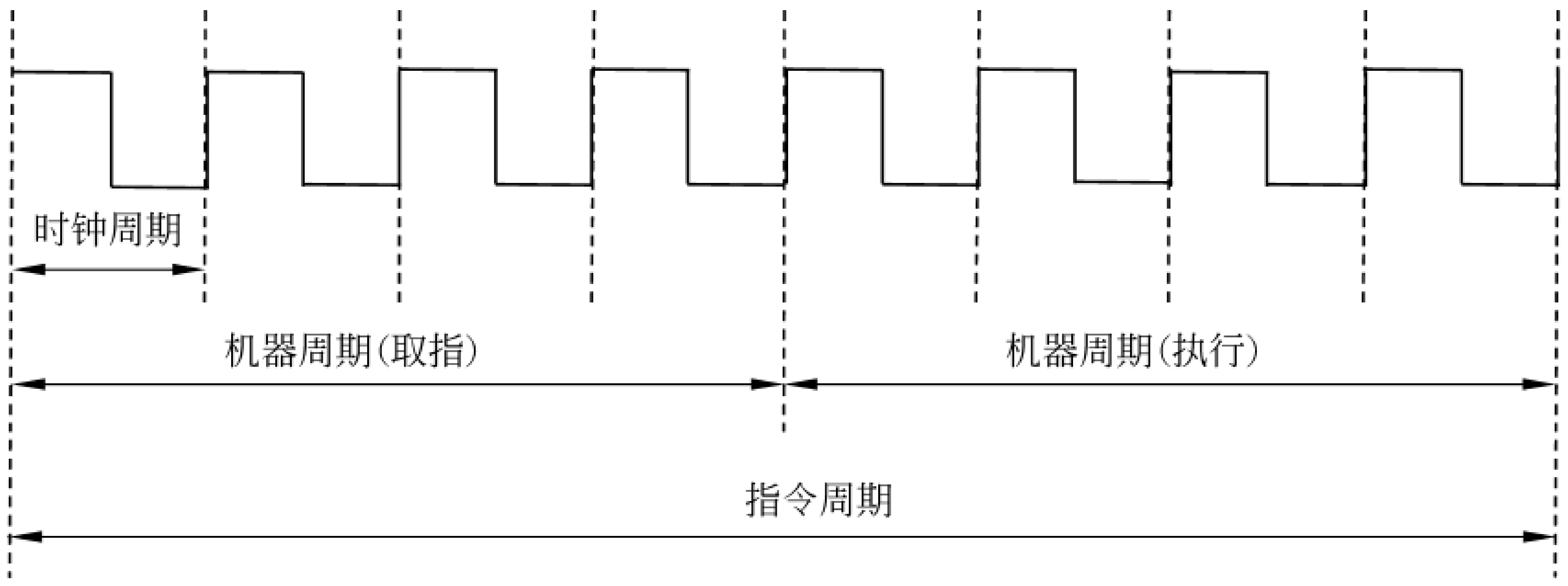


图 1-17　指令周期、机器周期和时钟周期关系

在计算机执行指令的过程中,计算机内各部件的每一个动作都必须严格遵守时间规定,这些部件的协调是用时序信号来控制的,时序信号由时序发生器来产生。时序发生器是产生控制指令周期的时序信号的部件,当 CPU 开始取指令并执行指令时,操作控制器利用时序发生器产生的定时脉冲的顺序和不同的脉冲间隔,提供计算机各部件工作所需的各种定时控制信号,有条理、有节奏地指挥机器各个部件按规定时间动作。

1.3　嵌入式系统概述

当前,计算机技术已经进入后 PC 时代。人们使用的计算机系统已经不再只是工作于传统的桌面 PC,很多的应用系统更多地工作在各种嵌入式平台上,如手机、智能洗衣机、智能手表以及数据采集器、智能定位装置和实时图像处理等各种嵌入式系统。

嵌入式系统是一种嵌入受控器件内部,为特定应用而设计的专用计算机系统。例如,智能洗衣机的智能洗涤程序及其平台构成的一个典型的嵌入式系统,该系统嵌入到洗衣机中并控制洗衣机的工作。区别于个人计算机系统,嵌入式系统通常执行的是带有特定要求和环境约束的任务,如要求极快的系统响应时间、较高的工作温度、极低的系统能耗和特别的安全性能。例如,应用于炼钢生产的天车定位装置,因为天车的快速移动要求具有极快的系统响应时间,而钢水辐射要求设备能工作在较高的环境温度。但是,个人计算机系统大多是通用计算机,具有通用的软硬件平台并安装有常用的应用软件,如现在大多数 PC 系统都是基于 Wintel 架构,即 Microsoft 公司的 Windows 操作系统与 Intel 公司的 CPU 所组成的个人计算机系统。

嵌入式系统一般针对一项特殊的任务,系统设计人员能够根据需要选择合适的硬件和软件平台,也可以减小系统尺寸或降低系统成本。实际上,很多嵌入式系统都会大量生产并广泛应用,嵌入式系统的成本就成为一个关键的设计问题。例如手机作为一种嵌入式系统,其价格作为未来市场推广成功与否一个关键要素。

嵌入式系统没有公认的明确定义,通常意义上,嵌入式系统可以描述为:以计算机技术为基础,面向特定应用,软件和硬件均可根据需要进行定制和裁剪,在系统可靠性、成本和功

耗等方面有着严格要求的专用计算机系统。

通用计算机系统和嵌入式系统相比较，嵌入式系统具有以下的显著特点：

1）专用性强

嵌入式系统大多面向特定的应用。因此，对嵌入式系统的硬件和软件系统都必须进行高效的设计和开发，尽可能去除不必要的冗余，保证面向应用的最为合适的运算能力。而且，在功耗、配置、内核处理能力、外围电路选择、系统响应时间和系统可靠性等方面具有明显的要求。例如，手机和智能手表等消费类产品要求具有良好的图形处理能力并有一定的容错能力，而航天军工和工业控制等工业类产品要求具有良好的实时计算能力且不允许出现错误，对系统的可靠性要求极高，也同时要求具有快速的系统响应。

2）实时性高

嵌入式系统对实时任务有很强的支持能力，能完成多任务并且有较短的中断响应时间。一般来说，需要优化中断控制器以及实时操作系统任务调度保障实时性。

3）种类繁多

嵌入式系统的多样性，集中体现为嵌入式微处理器和操作系统两个方面。

从嵌入式微处理器角度来说，已知的嵌入式微处理器大约有1000多种。嵌入式微处理器由通用计算机中的微处理器发展而来。与通用计算机的微处理器不同的是，在实际嵌入式应用中，只保留和嵌入式应用紧密相关的功能硬件，去除其他的冗余功能部分，因此其体积小、重量轻、功耗低、成本低及可靠性高。同时，嵌入式微处理器把CPU、ROM、RAM及I/O等元件以及各种外设集成到同一个芯片上，也称为单片机或微控制器。对于通用计算机系统来说，芯片基本上由Intel和AMD几家公司垄断，以X86和AMD64架构为主，操作系统软件方面，Microsoft公司的Windows几乎占据了90%的市场，可以说近代的通用计算机系统就是Wintel架构的垄断系统。但与全球PC市场不同的是，嵌入式微控制器约超过1000多种，体系架构有ARM、MIPS、PowrPC、X86、68K等30多个系列，以32位的嵌入式微控制器为例，Frescale、TI、ST、AVR、Atmel等公司就有100种以上的嵌入式微控制器。从目前市场看，ARM 32位微控制器架构占据垄断地位。

4）开发环境复杂

传统的通用计算机系统的开发环境与运行环境基本一致。例如，在一台计算机的Windows操作系统中开发一个学生管理系统，可能运行在另一台计算机的Windows操作系统上。

而对于嵌入式系统的开发来说，一般使用交叉开发模式，即将通用计算机与目标嵌入式系统进行连接，在通用计算机搭建开发环境，所开发的嵌入式代码“下载”到目标嵌入式系统进行运行。这种交叉开发模式无疑增加了系统开发的难度。

5）成本极其敏感

由于嵌入式系统往往大规模应用于各种生产、监测或消费等环境，往往使用量巨大。例如，遥控器、点菜机、温度监测设备、噪声监测设备甚至最近流行的智能手表、智能血糖仪等，都会有非常庞大的使用量。因此，每一个部件的成本都非常关键，总体成本或价格因素往往决定着产品的推广使用。

1.4　嵌入式系统架构

与通用计算机系统类似，嵌入式系统包括嵌入式硬件平台、嵌入式操作系统和嵌入式应用软件，如图1-18所示。

对于嵌入式系统来说，软件大多存储在只读存储器(ROM)中，一般不需要辅助存储器。

由于嵌入式系统面对的硬件种类多样，将开发者从繁琐的硬件细节中解放出来一直是嵌入式系统工业界面临的挑战。硬件抽象层(Hardware Abstraction Layer，HAL)是嵌入式系统开发中的一个重要中间件，硬件抽象层将微处理器的底层操作进行封装，隐藏了特定平台的硬件接口细节，为操作系统和应用软件提供虚拟硬件平台，使其具有硬件无关性，可在多种平台上进行移植。带有硬件抽象层的嵌入式系统结构，如图1-19所示。硬件抽象层的出现大大改进了嵌入式操作系统的通用性，硬件抽象层的定义一般由微处理器的生产厂家提供。

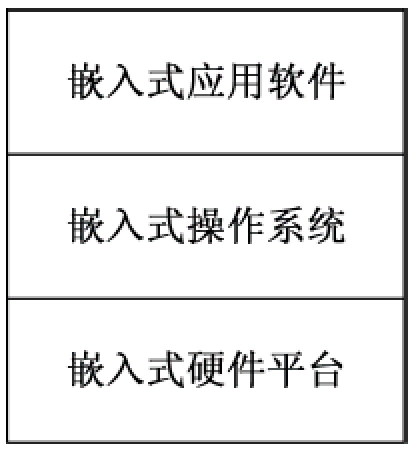


图1-18　嵌入式系统组成

图1-19　带硬件抽象层的嵌入式系统

板级支持包(Board Support Package，BSP)是介于主板硬件和操作系统中驱动层程序之间的一层，一般认为它属于操作系统一部分，主要是实现对操作系统的支持，为上层的驱动程序提供访问硬件设备寄存器的函数包，使之能够更好地运行于硬件主板。BSP是相对于操作系统而言的，不同的操作系统对应于不同定义形式的BSP，例如VxWorks的BSP和Linux的BSP相对于某一CPU来说尽管实现的功能一样，可是写法和接口定义是完全不同的。BSP主要功能是屏蔽硬件细节，提供硬件驱动，具体功能包括：

(1) 硬件初始化，主要是CPU的初始化，为整个软件系统提供底层硬件支持；

(2) 为操作系统提供设备驱动程序和系统中断服务程序；

(3) 定制操作系统的功能，为软件系统提供一个实时多任务的运行环境；

(4) 初始化操作系统，为操作系统的正常运行做好准备。

嵌入式操作系统(Embedded Operating System，EOS)是指用于嵌入式系统的操作系统。嵌入式操作系统是一种用途广泛的系统软件，通常包括与硬件相关的底层驱动软件、系统内核、设备驱动接口、通信协议、图形界面、标准化浏览器等。嵌入式操作系统负责嵌入式系统的全部软、硬件资源的分配、任务调度，控制、协调并发活动。与PC操作系统相比，具有可剪裁、体积小、强调实时性、与应用程序紧密耦合等特点。目前在嵌入式领域广泛使用

的操作系统有：嵌入式实时操作系统 μC/OS-Ⅱ、嵌入式 Linux、Windows Embedded、VxWorks 等，以及应用在智能手机和平板电脑的 Android、iOS 等。在很多简单应用中，一般不需要操作系统，应用程序直接通过硬件抽象层对硬件资源进行控制。

如图 1-20 所示，嵌入式系统的硬件层中包含嵌入式微控制器、外扩存储器（SDRAM、ROM、Flash 等）、设备 I/O 接口等。在一片嵌入式微控制器基础上添加电源电路、时钟电路和存储器电路，就构成了一个嵌入式核心控制模块，操作系统和应用程序都可以固化在 ROM 中。

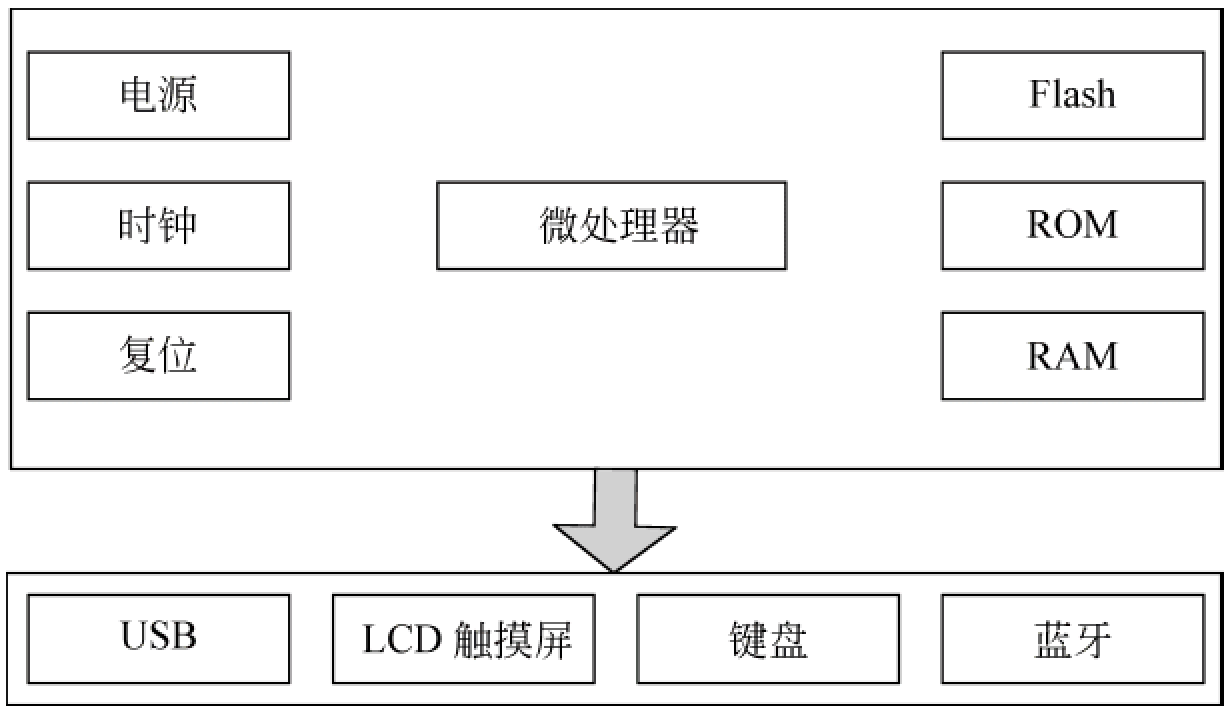


图 1-20　嵌入式系统的硬件组成

嵌入式微控制器是嵌入式系统的核心部件，一般采用 RISC 架构，嵌入式微控制器虽然在功能上和通用计算机的微处理器基本一致，但在可靠性、能耗、工作环境温度和抗电磁干扰等方面做了很多增强内部集成了 ROM、RAM 和大量外设接口，对于简单的应用，无需外扩存储，单片微控制器即可构成最小系统。

1.5　嵌入式系统的典型应用

1）工业控制

基于嵌入式芯片的工业自动化设备已经大量地应用于实际生产。很多 8 位、16 位和 32 位嵌入式微控制器应用于工业过程监控，如工业生产过程控制、数字机床、电力系统、电网设备监测和石油化工生产等，大大地减少了人力资源投入。就传统的工业控制产品而言，低端型采用的往往是 8 位单片机。随着嵌入式技术的发展，32 位甚至 64 位的微处理器逐渐成为工业控制设备的核心，在未来几年内必将获得长足的发展。

一款基于 ARM9 内核和 Windows CE 操作系统的工业计算机，如图 1-21 所示。该产品以 ARM9 低功耗嵌入式 CPU 为核心，主频为 400MHz，嵌入式操作系统采用 Windows CE 6.0，提供高性能嵌入式人机界面。

2）汽车电子

嵌入式系统在汽车和交通行业的应用主要表现为车辆导航、流量控制、信息监测和汽车

服务等。内嵌 GPS/北斗和 GSM、3G/4G 模块的移动定位终端已经在各种运输行业获得了成功的使用。由于 GPS 设备价格低廉,已经从尖端产品进入了普通百姓的家庭。一款基于 ARM11 内核的导航仪,如图 1-22 所示。该导航仪拥有分辨率 800×480 的 7 英寸高清数字屏,采用 ARM11 内核,主频 600MHz,内置 GPS,标配 4G SD(Secure Digital Memory Card)卡,支持 16G U 盘、32G SD 卡及硬盘扩展。

图 1-21　基于 ARM9 内核和 Windows CE 的工业计算机

图 1-22　基于 ARM11 内核的导航仪

3) 信息家电

信息家电是嵌入式系统最大的应用领域。冰箱、空调、洗衣机和电饭煲等家用电器的网络化和智能化将引领人们的生活步入一个崭新的空间。即使用户不在设备身边,也可通过网络进行远程控制。一款基于 ARM Cortex 内核和 Android 操作系统的电视盒,如图 1-23 所示。该款电视盒采用当今移动互联设备上 ARM Cortex-A9 1.2 GHz 核心处理器,内置 3D 图形处理器 Mali-400,搭配 512MB DDR3(Double Data Rate)大容量内存,性能超强,相当于通用计算机的高速运算能力,具有流畅的高清视频和游戏体验。同时采用 Google Android 4.0 操作系统,不但在系统稳定性有了进一步的保证,还可以随意安装使用 Google Market 数百万计的应用程序和游戏。

图 1-23　基于 ARM Cortex 内核和 Android 操作系统的电视盒

4) 环境监测

在很多环境恶劣、地况复杂的地区,嵌入式系统将实现无人值守 24 小时不间断监测,如水文资料实时监测、水土质量监测、地震监测、实时气象信息监测、水源和空气污染监测等。

一款基于 ARM 内核的物联网监测系统,如图 1-24 所示。搭配各种传感器的 ZigBee 节点用于测量温度、湿度、光照或气体成分等,ARM9 内核的通信网关支持 RS-232、USB 和

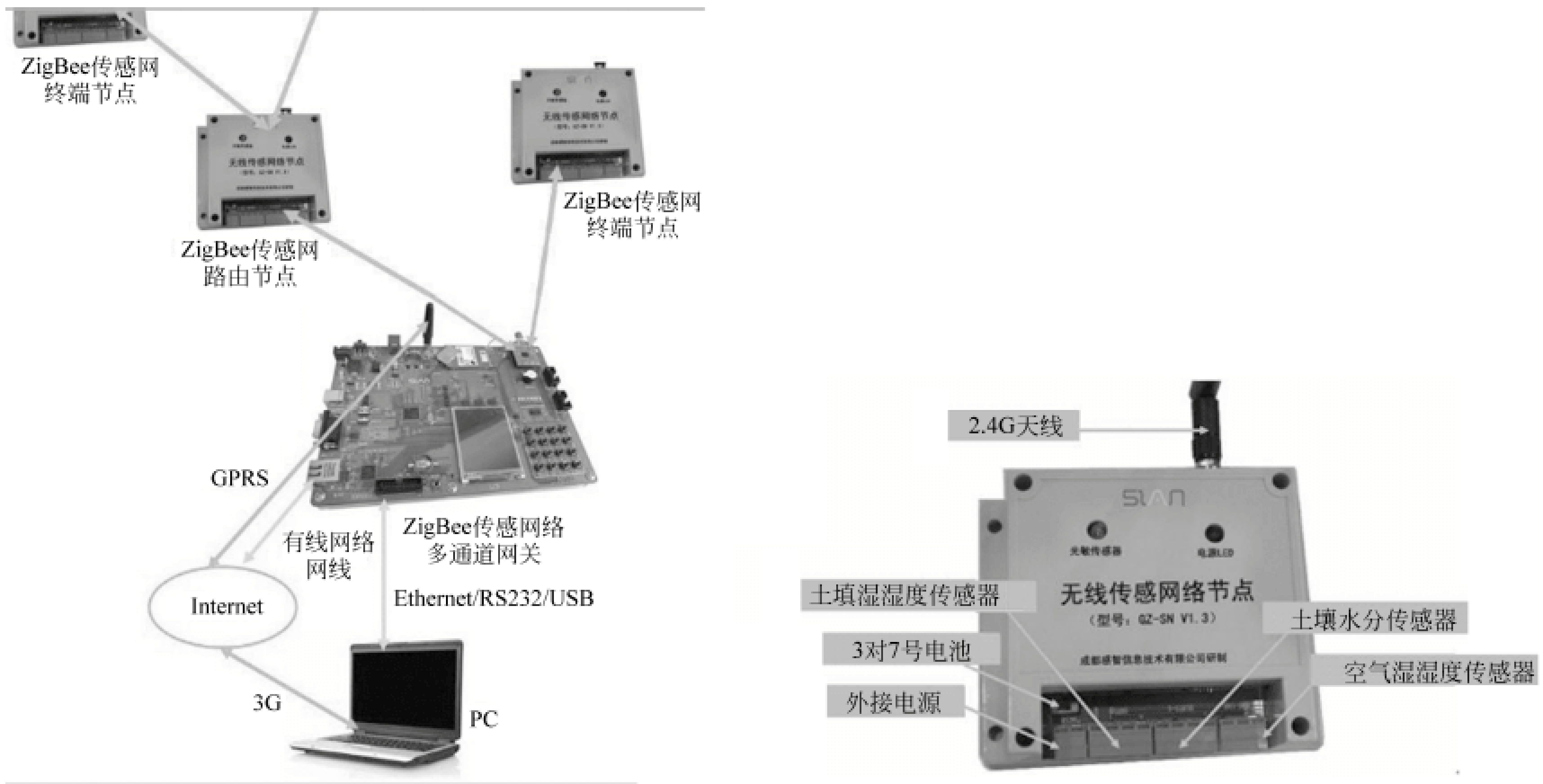


图 1-24　基于 ARM 内核的物联网监测系统

GPRS 等各种通信方式与上位机监控系统进行互联。

5）健康管理

嵌入式系统已经应用到健康管理的方方面面，如智能血压计、智能脉搏计和智能血糖仪等。嵌入式系统在健康管理方面的应用，必将成为未来最具有前景的应用领域之一。一款支持 iOS 或 Android 操作系统智能血压计，如图 1-25 所示。智能血压计可通过蓝牙连接智能手机传输数据，并生成图表，让用户更好地了解自己的血压状况。该智能血压计设计十分简洁，支持 iOS 及 Android 设置。不需要任何专业的医疗技巧，只需将尼龙腕带绑在手臂上，单击开始按键，就可以监测血压。

图 1-25　支持 iOS 或 Android 操作系统智能血压计

6）机器人

嵌入式芯片的发展将使机器人在微型化、智能化方面的优势更加明显，同时会大幅度降低机器人的价格，使其在工业领域和服务领域获得更为广泛的应用。一款基于 ARM 内核和 μC/OS-Ⅱ操作系统的家庭机器人，如图 1-26 所示。该机器人集红外、超声波、湿度、温度、烟雾和煤气等多种传感器于一体，具有对外无线通信的功能，使用了嵌入式领域应用广

泛、处理能力极强的 32 位 ARM 处理器，软件采用实时性操作系统 μC/OS-Ⅱ，来协调机器人自身复杂的机械控制及处理周围复杂未知的环境因素。

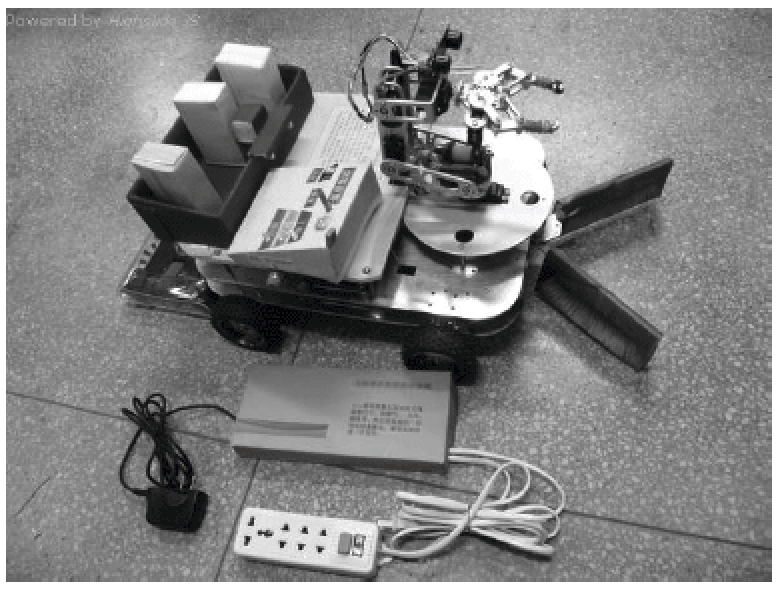


图 1-26　基于 ARM 内核和 μC/OS-Ⅱ操作系统的家庭机器人

1.6　典型嵌入式开源硬件和软件系统

1.6.1　开源硬件平台

由于嵌入式产品都有相似的微处理器内核及通用外围功能单元，将它们集合起来，可做成一个供众多嵌入式产品个性化开发的产品平台。最引人注目的是由树莓派(Raspberry Pi)引发的智能化通用板级开源硬件。

最早推出的 Raspberry Pi 是为学习计算机编程而设计的一个只有信用卡大小的板级微型电脑，配置了 Linux 操作系统。由于低价位、功能强大、有众多的外围电路与 I/O 端口、易开发的软件配置，树莓派迅速成为板级开源硬件的理想化通用产品平台。到目前为止，已出现了香蕉派、Arduino、BeagleBoard 等一系列开源硬件平台，其中三大主流平台位 Arduino、BeagleBoard 和 Raspberry Pi，它们已建立了完整的硬件、软件生态。

1) 树莓派

树莓派(Raspberry Pi)由英国的树莓派基金会所开发，目的是以低价硬件及自由软件刺激学校的基础计算机科学教育，如图 1-27 所示。树莓派配备一枚 700MHz 博通出产的 ARM 架构 BCM2835 处理器，256MB 内存(B 型已升级到 512MB 内存)，使用 SD 卡当作存储媒体，且拥有一个 Ethernet，两个 USB 接口，以及 HDMI(支持声音输出)和 RCA 端子输出支持。操作系统采用开源的 Linux 系统，比如 Debian、ArchLinux，自带的 Iceweasel、KOffice 等软件能够满足基本的网络浏览、文字处理以及计算机学习的需要。树莓派基金会提供了基于 ARM 架构的 Debian、Arch Linux 和 Fedora 等的发行版供大众下载，以 Python 作为主要编程语言，支持 BBC BASIC、C 语言和 Perl 等编程语言。树莓派基金会于 2016 年 2 月发布了树莓派 3，较前一代树莓派 2，树莓派 3 的处理器升级为了 64 位的博通 BCM2837，并首次加入了 Wi-Fi 无线网络及蓝牙功能，而售价仍然是 35 美元，目前已发布

树莓派 4。

2）Arduino

Arduino 是一个开放源代码的单芯片微计算机，由一个欧洲开发团队于 2005 年冬季开发。它使用了 Atmel AVR 单片机，采用了基于开放源代码的软硬件平台，构建于开放源代码 simple I/O 接口板，并且具有使用类似 Java、C 语言的 Processing/Wiring 开发环境。

Arduino 能通过各种各样的传感器来感知环境，通过控制灯光、马达和其他的装置来反馈、影响环境。Arduino 包括一个硬件平台（Arduino Board）和一个开发工具（Arduino IDE）。两者都是开放的，既可以获得 Arduino 开发板的电路图，也可以获得 Arduino IDE 的源代码。Arduino Board 提供了基本的接口和 USB 转串口模块，如图 1-28 所示。使用者只需要用一个 USB 线就可以连接计算机和 Arduino Board，完成编程和调试。Arduino 使用一种简单的专用编程语言，使用者不必掌握汇编语言和 C 语言等复杂技术就可以进行开发。IDE 可免费下载，并开放源代码，跨平台，极为便利。

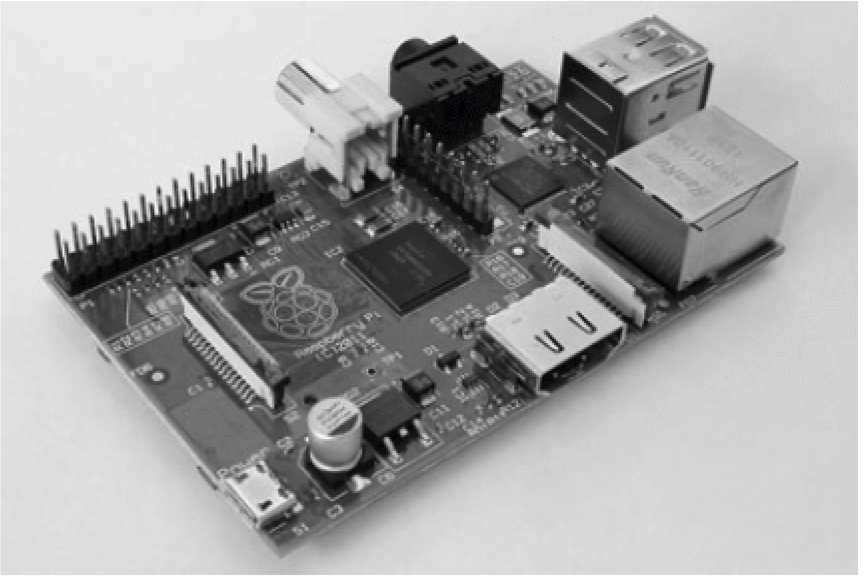


图 1-27　树莓派 B2

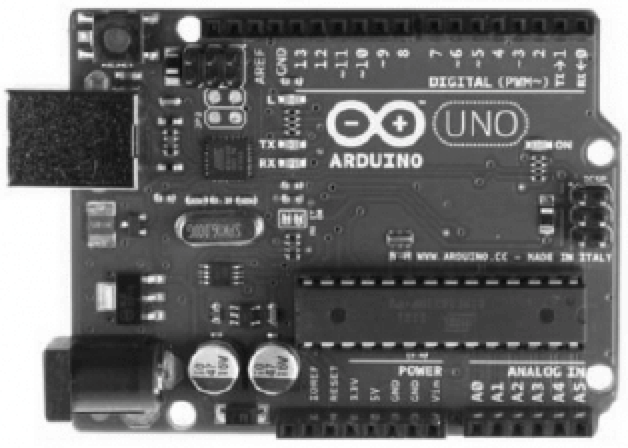


图 1-28　Arnduino 开发板

3）BeagleBoard

BeagleBoard 是开源硬件领域知名社区 BeagleBoard. org 推出的、全球第一款开源的 ARM 开发板，如图 1-29 所示。不同于热门的开源平台 Arduino，BeagleBoard 的功能更强大、应用更复杂。BeagleBoard 跨越了台式机和嵌入式计算机的界限，同时与开源社区展开创建全新应用的协作，为开源社区提供成本更低、更新、更出色的开发平台。BeagleBone 是 BeagleBoard 的升级版本，只集成了一些必不可少的接口功能，如 USB、以太网口。继 BeagleBone 之后，德州仪器推出的 BeagleBone Black，采用 TI 最新 Cortex-A8 架构 Sitara 处理器，主频可提升至 1GHz。

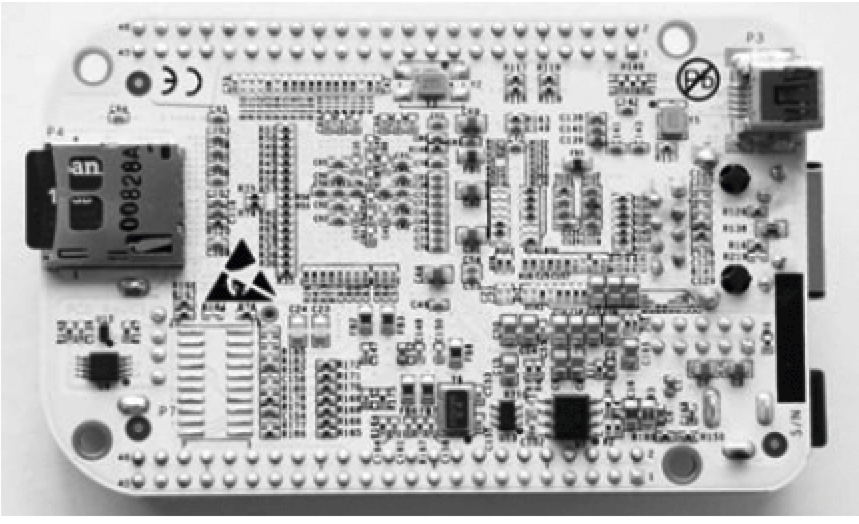


图 1-29　BeagleBone

1.6.2　嵌入式开源操作系统

嵌入式操作系统是嵌入式应用系统的核心软件平台，目前常见的嵌入式操作系统有：eCos、μC/OS、VxWorks、pSOS、Nucleus、ThreadX、Rtems、QNX、INTEGRITY、OSE、C Executive、CMX、SMX、emOS、Chrous、VRTX、RTX、FreeRTOS、LynxOS、ITRON、Symbian、RT-thread，以及 Linux 家族的各种版本(如 μClinux、Android 等)，还有微软家族的 WinCE、Windows Embedded、Windows Mobile 等。

1) Linux

Linux Torvalds 在 1991 年发表的 Linux 开放操作系统，是由互联网上的志愿者们开发的，它吸引了许许多多忠实的追随者。自 1999 年稳定的 2.2 版本发布以来，Linux 不仅已经在服务器和台式机上取得了巨大的成功，也正在嵌入式系统中大放异彩。许多人认为，Linux 之所以获得嵌入式市场的广泛认可，关键是得益于 Linux 极高的质量和极强的生命力。当然，能够给 Linux 开发人员提供充分的灵活性和开放的源码，不收取运行许可使用费也是开发者选择 Linux 的极好理由。与商业软件授权方式不同的是，开发者可以自由地修改 Linux，能最大地满足他们的应用需要。在技术上，因为基于 UNIX 技术，Linux 提供广泛的功能强大的操作系统功能，包括内存保护、进程和线程，以及丰富的网络协议。Linux 与 POSIX 标准兼容，从而提高了应用的可移植性。Linux 支持多种微处理器、总线架构和设备，通常情况下，芯片公司的驱动程序、应用相关的中间件、工具和应用程序都是先为 Linux 开发，后来才移植到其他 OS 平台的。这些特性都非常适合于嵌入式系统应用。

2) MontaVista Linux

MontaVista Linux 不只是一个通用的 Linux 发行版，更是为嵌入式系统所需的可靠性和实时性(通过对 2.4 内核加入实时补丁)而精心设计的，支持高端嵌入式系统使用的处理器架构 x86、ARM、PowerPC 和 MIPS，以及一系列的驱动程序和板级支持包。它有一整套的开发工具、闪存和固态存储文件系统，还有很容易监视系统完整性和性能的各种工具。MontaVista Linux 在通信基础设备、智能手机、数字电视机和机顶盒等各种嵌入式系统中得到广泛应用。

3) eCos

eCos 全称是 Embedded Configurable Operating System，它诞生于 1997 年，eCos 最大的特点是模块化，内核可配置。它是一个针对 16/32/64 位处理器的可移植开放源代码的嵌入式 RTOS。eCos 提供的 Linux 兼容的 API 能让开发人员轻松地将 Linux 应用移植到 eCos。eCos 的核心具备一般 OS 功能，如驱动和内存管理、异常和中断处理、线程的支持，还具备实时操作系统的特点，如可抢占、最小中断延迟、线程同步等。eCos 支持大量外设、通信协议和中间件，如以太网、USB、IPv4/IPv6、SNMP、HTTP 等。

4) Android

Android 是谷歌公司开发的一个针对高端智能手机的操作系统。其实 Android 不仅仅是一个操作系统，也是一个软件平台，可以应用在更加广泛的设备中。在实际应用中，

Android 是一个在 Linux 上的应用架构，优势是能够帮助开发者快速地布置应用软件。Android 的开发主要还是集中在移动终端上，在其他的市场上 Android 也潜力巨大。比如智能电视、消费电子产品、通信、汽车电子产品、医疗仪器和智能家居应用等。

5) μC/OS-Ⅱ

μC/OS-Ⅱ由 Micrium 公司提供，是一个可移植、可固化、可裁剪、抢占式多任务实时内核，它适用于多种微处理器、微控制器和数字处理芯片(已经移植到超过 100 种以上的微处理器应用中)。该系统源代码开放、整洁、注释详尽。μC/OS-Ⅱ可管理多达 63 个应用任务，并可以提供如下服务：信号量、互斥信号量、事件标识、消息邮箱、消息队列、任务管理、固定大小内存块管理、时间管理另外，在 μC/OS-Ⅱ内核之上还可以选增、μC/FS 文件系统模块、μC/GUI 图形软件模块、μC/TCP-IP 协议栈模块、μC/USB 协议栈模块等。

6) μCLinux

μCLinux 表示 micro-control Linux. 即"微控制器领域中的 Linux 系统"，是 Lineo 公司的主打产品，同时也是开放源码的嵌入式 Linux 的典范之作。μCLinux 主要是针对目标处理器没有存储管理单元 MMU(Memory Management Unit)的嵌入式系统而设计的，是唯一可以在低端 MCU 上运行的 Linux，可以在特定的 Cortex-M3、M4 和 M7 等型号上运行。μClinux 的 RAM 和 ROM 资源需求较多，需要 MCU 内置存储器控制器，使用外部扩展 DRAM 芯片来满足内存要求。现在 μClinux 已被并入到主线 Linux 内核中。

7) FreeRTOS

FreeRTOS 这是一个开源的项目，属于轻量级内核，API 比较全，支持 AVR、ARM、MSP430 等处理器，同时有移植好的 TCP/IP 协议栈 μIP。

8) ARM Mbed

ARM 面向物联网的操作系统针对小巧、电池供电的物联网端点，这些端点在 Cortex-M 系列 MCU 上运行，可能只有 8KB 内存。mbend 提供了多线程和实时操作系统支持，在设计当初就针对无线通信，可通过 Mbed Device Connector 来安全地提取数据的云服务。

除以上开源嵌入式操作系统以外，微软的 Windows CE、Windows Phone、苹果的 iOS 都是典型主流嵌入式操作系统，老牌的嵌入式操作系统代表为风河公司的 VxWorks。

VxWorks 是由支持多核、32/64 位嵌入式处理器、内存管理的 Vxworks、workbench 开发工具(包括多种 C/C++ 编译器和调试器)、连接组件(USB、IPv4/IPv6、多种文件系统等)、网络协议和图像多媒体等模块组成。除了通用平台外，VxWorks 还包括支持工业、网络、医疗和消费电子等的特定平台产品。风河公司的 VxWorks 以其高可靠性和优异的实时性被广泛应用在通信、军事、航空航天、工业控制等领域。比如在美国的 F-16、FA-18 战斗机、B-2 隐形轰炸机和爱国者导弹上都有使用，最为著名的是 1997 年 4 月在火星表面登陆的火星探测器、2008 年 5 月登陆的凤凰号和 2012 年 8 月登陆的好奇号火星车，也都使用到了 VxWorks。

Chapter 2

第 2 章　Cortex-M3 微处理器的体系结构

【导读】 ARM Cortex-M3 是目前低成本嵌入式系统使用最为广泛的 CPU 内核，本章首先回顾 ARM 微处理器的发展，重点介绍 Cortex-M3 处理器的内核结构、工作模式、存储映射等基本原理，然后以 ST 公司的 STM32L152 为例，对该微控制器的结构、引脚说明、时钟控制、存储等进行详细介绍，为指令系统和后续 I/O 接口的学习奠定基础。

2.1　ARM 微处理器系列介绍

ARM 的全拼是 Advanced RISC Machines，中文的意思为先进的精简指令集机器，是一家总部位于英国剑桥的半导体微处理器公司。目前，ARM 在手机处理器市场占据了超过 90%的份额，在平板电脑处理器市场占据了超过 80%的份额。

表 2-1 为 ARM 的系列处理器代号及其处理器架构版本。ARM 处理器大约有 6 个流行的产品系列，分别为 ARM7、ARM9、ARM10、ARM11、SecureCore 和 Cortex。其中，ARM7、ARM9、ARM10 和 ARM11 是早期的处理器命名方式，每个系列提供可配置的不同性能的处理器版本；SecureCore 系列主要面向安全设备设计；ARM11 之后，所有处理器均以 Cortex 系列命名，根据不用的市场包括三个子系列，分别是 Cortex-A、Cortex-M 和 Cortex-R。其中，Cortex-A 系列面向复杂操作系统和用户应用，支持 ARM、Thumb 和 Thumb-2 指令集；Cortex-R 系列面向嵌入式或实时系统，也支持 ARM、Thumb 和 Thumb-2 指令集；Cortex-M 系列面向成本敏感的嵌入式应用，只支持 Thumb-2 指令集。

ARM 系列处理器有三种指令集：

(1) ARM 指令集：ARM 指令集为 32 位指令集，可以实现 ARM 架构下的所有功能。

(2) Thumb 指令集：Thumb 指令集是针对代码存储密度的需求对 32 位 ARM 指令集的改进，其目标是实现更高的代码密度。具体做法是将 32 位 ARM 指令集的部分指令压缩成 16 位的编码方式，而当指令执行时，再解码成相应的 32 位 ARM 指令功能。这种经过压缩-解码方式，以牺牲部分处理器性能换取了代码密度的提升。相对于 ARM 指令集，Thumb 指令集在代码密度方面大约提升了 30%，但 Thumb 不是一个完整的指令集，部分 32 位指令无法压缩，Thumb 需要和 ARM 指令集配合使用，两种指令执行时处理器需要在不同的状态切换。

(3) Thumb-2 指令集：Thumb-2 指令集是在 Thumb 指令集的基础上发展而来 16 位/

32 位混合指令集，增加了一些 16 位 Thumb 指令来改进程序的执行流程，增加一些新的 32 位 Thumb 指令来实现 ARM 指令的专有功能，因此，Thumb-2 指令集中有两类不同长度的指令，不兼容 ARM 指令集，并且采用了新方法实现 16 位和 32 位指令的执行，无需进行 Thumb 和 ARM 指令的压缩解压转换，也无需在 ARM 和 Thumb 状态进行来回切换，大大提升了运算效率。

Cortex-M 系列是基于 ARMv7 架构。ARMv7 架构是在 ARMv6 架构的基础上发展而来。ARMv7 架构采用 Thumb-2 技术，Thumb-2 技术比纯 32 位代码少使用大约 31%的内存，却比基于 Thumb 技术的代码在性能上提高大约 38%。ARM 处理器代号与指令集架构之间的关系如表 2-1 所示。

表 2-1　ARM 处理器与对应的架构

ARM 处理器内核代号	架　构
ARM1	ARMv1
ARM2	ARMv2
ARM2As，ARM3	ARMv2a
ARM6，ARM600，ARM610，ARM7，ARM700，ARM710	ARMv3
StrongARM，ARM8，ARM810	ARMv4
ARM7TDMI，ARM710T，ARM720T，ARM740T，ARM9TDMI，ARM920T，ARM940T	ARMv5T
ARM9E-S，ARM10TDMI，ARM1020E	ARMv5TE
ARM1136J(F)-S，ARM1176JZ(F)-S，ARM11，MPCore	ARMv6
ARM1156T2(F)-S	ARMv6T2
ARM Cortex-M，ARM Cortex-R，ARM Cortex-A	ARMv7、ARMv8

第一款采用 ARMv7-M 架构的处理器架构是 Cortex-M3，随后 ARM 针对嵌入式市场的需求，形成了系列 M 处理器架构。

Cortex-M0、M0＋、M1 系列：Cortex-M0 是目前最小的 ARM 处理器，该处理器支持 ARMv6-M 架构，芯片面积非常小，能耗极低，且编程所需的代码占用量很少，这就使得开发人员可以直接跳过 16 位系统，以接近 8 位系统的成本开销获得 32 位系统的性能。Cortex-M0＋是以 Cortex-M0 处理器为基础，保留了全部指令集和数据兼容性，同时进一步降低了能耗，提高了性能，两级流水线，性能效率可达 1.08DMIPS/MHz。Cortex-M1 是第一个专为 FPGA 中的实现设计的 ARM 处理器。

Cortex-M3：ARMv7-M 架构，改为 3 级流水哈佛结构，是目前 M 系列中应用最广泛的处理器架构。Cortex-M4 在 Cortex-M3 基础上增加了 DSP 支持和单精度浮点运算加速，用以满足需要有效且易于使用的控制和信号处理功能混合的数字信号控制市场。Cortex-M7 具有六级流水线、灵活的系统和内存接口、缓存(Cache)、DSP 和双精度浮点运算加速。Cortex-M 系列核心的比较如表 2-2 所示。

表 2-2　Cortex-M 系列核心的对比

类　别	Cortex-M0	Cortex-M3	Cortex-M4	Cortex-M7
体系结构	ARMv6-M	ARMv7-M	ARMv7-M	ARMv7-M
ISA 指令集	Thumb，Thumb-2	Thumb，Thumb-2	Thumb，Thumb-2	Thumb，Thumb-2
DSP 扩展	—	—	支持	支持
浮点单元	—	—	单精度浮点	双精度浮点
DMISP 性能	0.9	1.25	1.25	2.5
内存保护	—	MPU	MPU	MPU

2.2　ARM Cortex-M3 体系结构

Cortex-M3 微处理器是一个高性能的 32 位处理器，主要面向微控制器市场。它集成了名为 CM3Core 的中央处理器内核和高效的总线，实现了内置的中断控制、存储器保护以及系统的调试和跟踪功能。具有快速中断处理机制、高效的内核运算性能和多种低功耗睡眠模式，支持 Thumb-2 指令集，确保较高的代码密度和较低的存储要求。

2.2.1　总体架构

Cortex-M3 处理器的内部架构如图 2-1 所示，其主要包括 5 个功能部件。

(1) 处理器内核 CM3Core，Cortex-M3 处理器的中央处理核心。

(2) 中断控制器(Nested Vectored Interrupt Controller，NVIC)。NVIC 是一个在 Cortex-M3 中内建的中断控制器，支持中断嵌套，采用向量中断机制，在中断发生时，它会自动取出对应的中断服务例程入口地址，并且直接调用，无需软件判定中断源，由此缩短了中断延时。

(3) 总线矩阵 BusMatrix，总线矩阵是 Cortex-M3 内部总线系统的核心，它是一个 AHB 总线互连网络，可以让数据在不同的总线之间并行传送(相当于网络交换机功能)。

(4) 存储保护单元 MPU(可选单元)，其主要功能是把存储器分成不同区域分别予以保护，让某些区域在用户模式下变成只读，特权模式下可读写，从而实现关键数据的保护。

(5) 处理器跟踪和调试接口，包括 FPB(Flash 修补和断点单元)、DWT(数据观察点和触发单元)、ITM(指令跟踪宏单元)、ETM(嵌入式跟踪宏单元)和 TPIU(跟踪端口接口单元)和一个串行线调试端口(SW-DP)/串口线 JTAG 调试端口(SWJ-DP)。ETM、TPIU、SW/JTAG-DP 和 ROM 表是可选的。

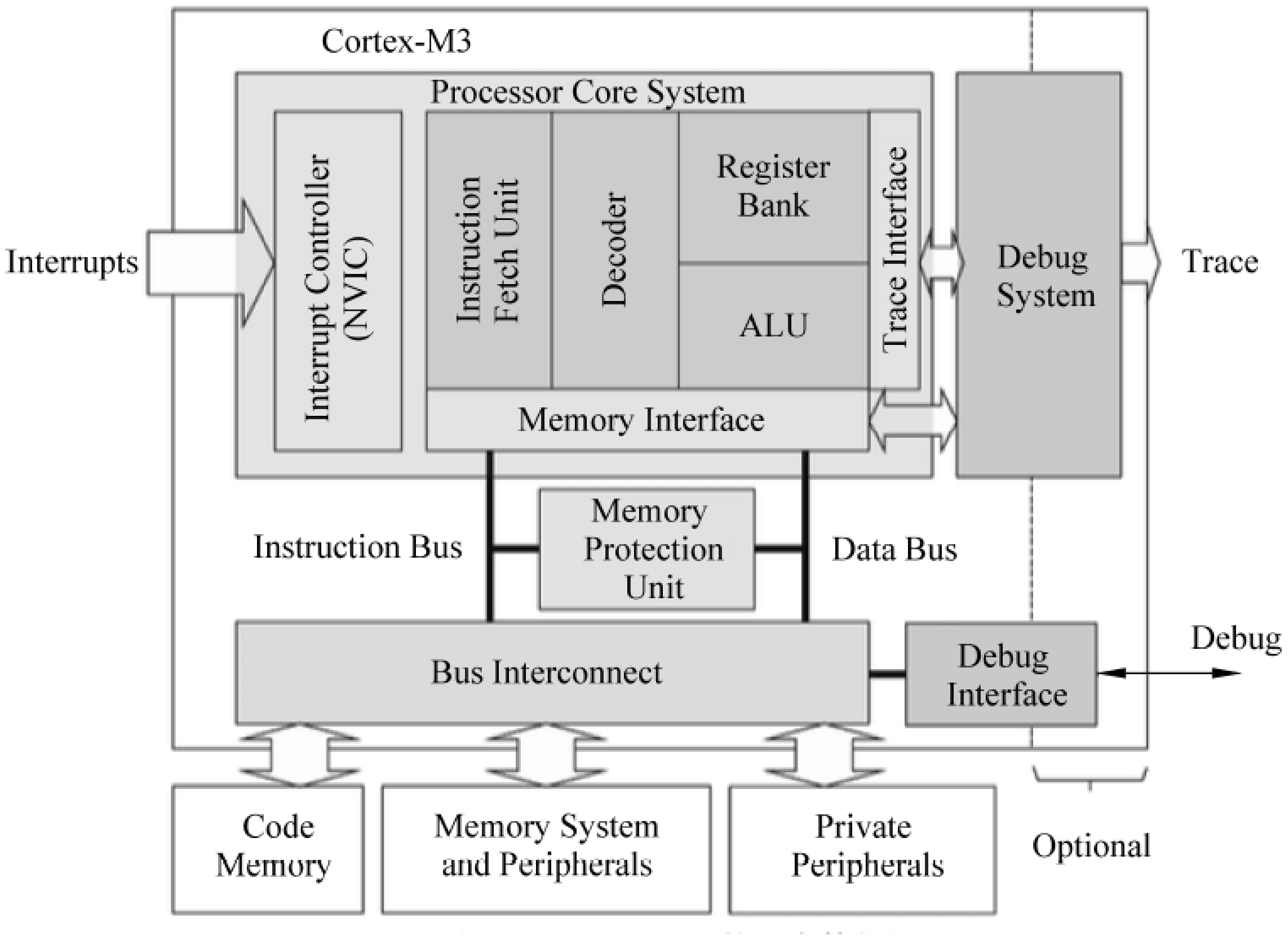


图 2-1 Cortex-M3 处理器内核架构

1. 处理器内核

Cortex-M3 中央内核采用哈佛架构，指令和数据各使用一条总线。内核流水线分 3 个阶段：取指、译码和执行。当遇到分支指令时，译码阶段也包含指令预取，提高执行速度。Cortex-M3 内核包含 Thumb 和 Thumb-2 指令译码器、支持硬件乘法和除法的 ALU、控制逻辑和用于连接处理器其他部件的接口。Cortex-M3 处理器是一个 32 位处理器，带有 32 位宽的数据路径，寄存器和存储器接口。其中有 13 个通用寄存器，两个栈指针，一个链接寄存器，一个程序计数器和一系列包含编程状态寄存器的特殊寄存器。Cortex-M3 处理器支持两种工作模式（线程模式和异常处理模式）和两个等级的访问形式（有特权或无特权），实现对操作系统安全保护的支持。

2. NVIC 中断控制器

Cortex-M3 集成了一个可配置的嵌套向量中断控制器 NVIC。NVIC 支持 256 个中断和 256 种中断优先级，与处理器核心紧密耦合，提供中断服务程序（Interrupt Service Routine，ISR）的快速执行。其主要特性包括：

（1）CPU 内部占用 16 个中断，提供给处理器厂家的外部中断可配置为 1～240 个，中断可屏蔽，同时也支持一个不可屏蔽中断（Non-Maskable Interrupt，NMI）。

（2）优先级的种类可配置，支持最少 8 种，最多 256 种不同的优先级，支持中断嵌套。

（3）支持优先级分组，中断优先级可动态地重新配置。

（4）支持末尾连锁（tail-chaining）和迟到（late arrival）中断技术，提高响应速度。

（5）处理器状态在进入中断时自动保存，在保存状态的同时从存储器中取出异常向量，实现更加快速地进入 ISR（中断服务程序），中断返回时自动恢复，无需多余的指令。

3. 总线矩阵

总线矩阵用于配置 Cortex-M3 处理器核心和系统总线、存储器以及调试单元之间的总线连接，其支持的总线包括：

(1) ICode 总线，该总线用于从代码空间取指令和向量，是 32 位 AHBLite 总线。

(2) DCode 总线，该总线用于对代码空间进行数据加载/存储以及调试访问，是 32 位 AHBLite 总线。

(3) AHB 系统总线，该总线用于对系统空间执行取指令和向量，数据加载/存储以及调试访问，是 32 位 AHBLite 总线。

(4) PPB 私有外围设备总线，该总线用于对 PPB 空间(处理器集成的调试单元)进行数据加载/存储以及调试访问，是 32 位 APB 总线。

2.2.2 操作模式

Cortex-M3 处理器有两种工作状态：

(1) Thumb-2 状态：Thumb-2 指令的正常执行状态。

(2) 调试状态：处理器停机调试时进入该状态。

Cortex-M3 处理器支持两种工作模式：线程模式和异常处理模式。

(1) 线程模式(Thread Mode)：处理器工作在线程模式，用于执行应用程序，在复位时处理器进入线程模式，异常返回时也会进入该模式。

(2) 异常处理模式(Handler Mode)：处理器工作在异常处理模式，用于处理异常事件和中断。出现异常时处理器进入异常处理模式，当完成了异常处理之后，处理器将返回到线程模式。

Cortex-M3 处理器的程序代码运行级别分为特权级执行和非特权级执行(也称用户级执行)。

非特权级别执行时对有些资源的访问受到限制或不允许访问(如 CPS 指令、系统控制空间的大部分寄存器等)，特权级别执行可以访问所有资源。当处理器工作在异常处理模式下时始终是特权访问，工作在线程模式下可以是特权访问，也可以是非特权访问。

Cortex-M3 在线程模式下，控制寄存器 CONTROL 用于决定处理器处于特权级别还是非特权级别。程序在特权级别下可通过 MSR 指令将 CONTROL 寄存器的最低位 CONTROL[0]置 0，配置为非特权(用户)访问，但在非特权访问级别，本身不能主动回到特权访问。

【思考题： 如何让用户级的程序主动进入特权级?**】**

根据 Cortex-M3 处理器的工作模式和特权等级，可以将 Cortex-M3 处理器划分为三种工作状态：特权级异常处理模式、特权级线程模式和非特权级线程模式，工作状态之间的转换，如图 2-2 所示。

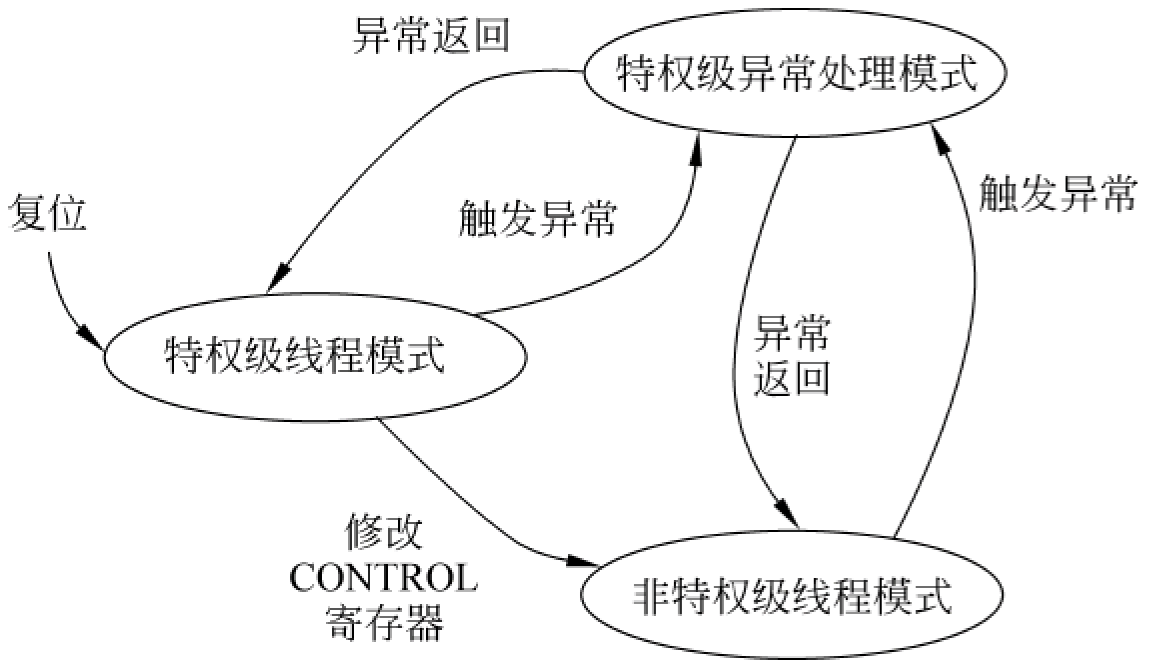


图 2-2　处理器工作状态的切换

由图 2-2 可见，只有处于特权级才可以通过改写控制寄存器 CONTROL 来改变处于线程模式的程序的特权等级，即可以由特权级线程模式直接过渡到非特权级线程模式，反之，则不能由非特权级线程模式通过修改控制寄存器 CONTROL 直接过渡到特权级线程模式。处于非特权线程模式的程序，只有进入异常处理模式才具有特权执行权限，此时可以通过修改 CONTROL 寄存器让程序在中断服务执行完成后回到特权级线程模式。

三种状态之间的切换示例如图 2-3 所示。

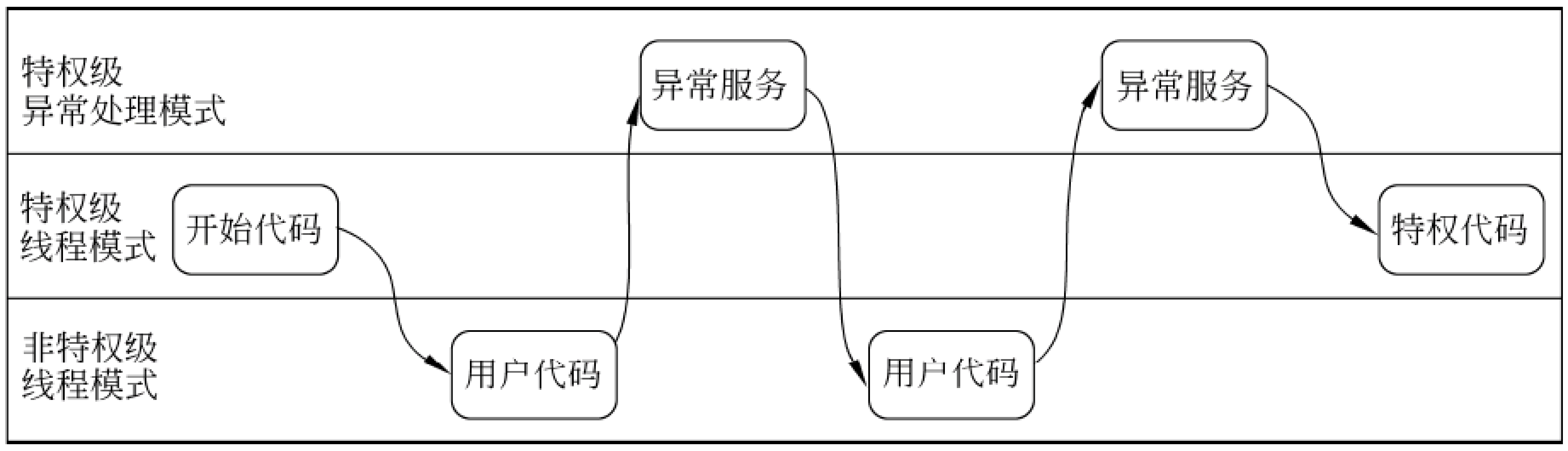


图 2-3　特权级别和非特权级别的转换示例

把代码按特权级和用户级分开，主要是操作系统的需求，操作系统工作在特权级，用户程序工作在非特权级，当用户代码出问题时，不会影响整个系统的运行。结合 MPU，可以防止用户代码访问不属于它的内存区域。

2.2.3　寄存器

Cortex-M3 处理器寄存器堆中寄存器如表 2-3 所示，包括了通用寄存器、特殊功能寄存器以及状态寄存器。不同的寄存器要求在不同特权等级下进行访问，例如，PRIMASK 优先权屏蔽寄存器只能在特权级下才能进行访问。

表 2-3　Cortex-M3 处理器中的寄存器

寄　存　器		访 问 类 型	要求的特权等级	重　置　值
R0～R12		RW	特权级或非特权级	不确定
R13	MSP	RW	特权级	不确定
	PSP	RW	特权级或非特权级	不确定
LR(R14)		RW	特权级或非特权级	0xFFFFFFFF
PC(R15)		RW	特权级或非特权级	不确定
PSR	APSR	RW	特权级或非特权级	不确定
	IPSR	RO	特权级	0x00000000
	EPSR	RO	特权级	0x01000000
PRIMASK		RW	特权级	0x00000000
FAULTMASK		RW	特权级	0x00000000
BASEEPRI		RW	特权级	0x00000000
CONTROL		RW	特权级	0x00000000

如图 2-4 所示，Cortex-M3 的 16 个通用寄存器 R0～R15，分为通用寄存器 R0～R12，特殊寄存器 R13～R15。

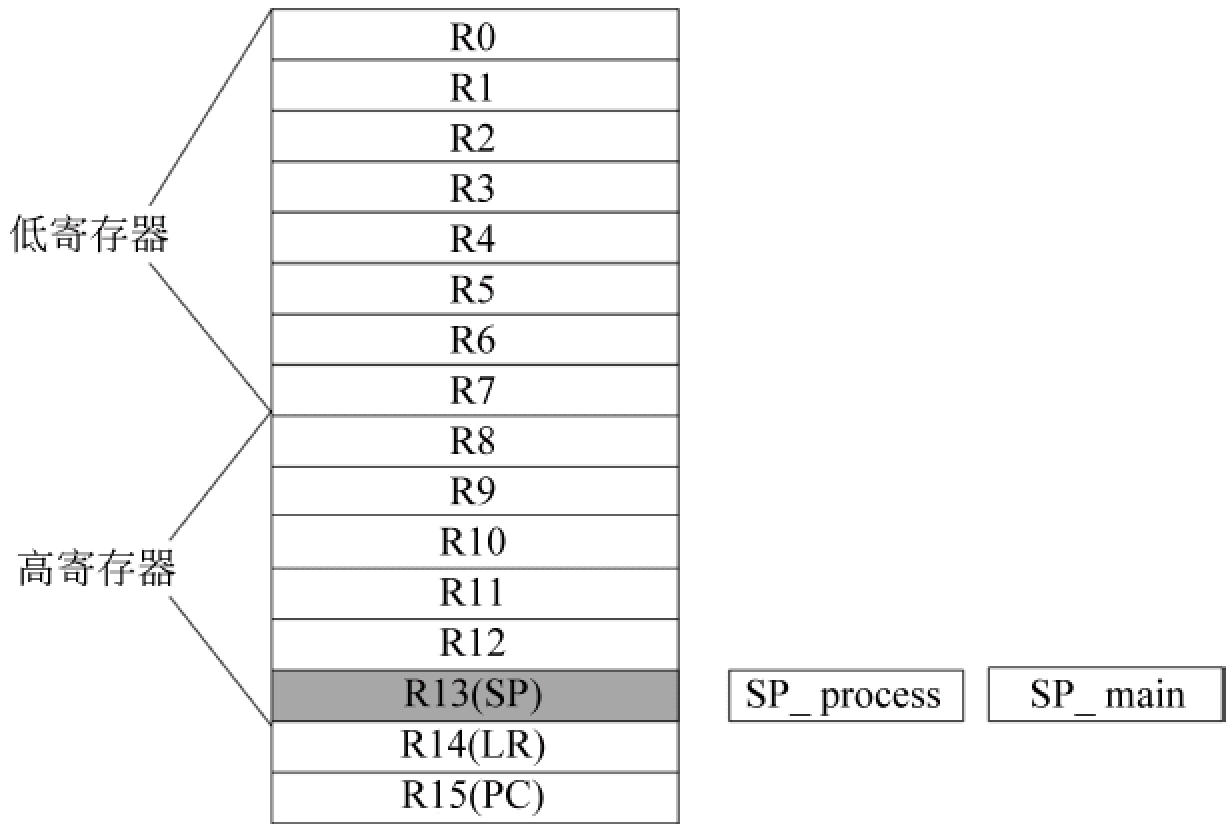


图 2-4　寄存器堆

1. 通用寄存器 R0～R12

通用寄存器 R0～R12 可以被大多数指令使用，按照 16 位和 32 位指令的划分，通用寄存器分为两段：

- 低组寄存器 R0～R7：可以被所有的指令访问。
- 高组寄存器 R8～R12：可以被所有 32 位指令访问，不能被 16 位指令访问。

【思考题：寄存器为何要分为两段?**】**

2. 栈指针寄存器 R13

为了避免操作系统的栈因应用程序的错误使用而毁坏，应用程序和操作系统的栈不共享栈指针，即 R13 寄存器(也记作 SP)在特权模式和用户模式下分别对应不同的栈指针寄存器，这两个栈指针在特权级别变化时自动切换。Cortex-M3 处理器的两个栈指针分别为：

(1) 主栈指针 SP_main，记作 MSP(Main Stack Pointer)。这是默认的栈指针，它由 OS 内核、异常服务例程以及所有需要特权访问的应用程序代码来使用。

(2) 进程栈指针 SP_process，记作 PSP(Process Stack Pointer)，用于应用程序代码。

异常处理模式下始终使用 MSP，而线程模式可配置为 MSP 或 PSP。处理器复位后，处于特权级线程模式，所有代码都使用主栈 MSP。异常处理模式，异常处理程序 ISR 可以通过改变其在退出时使用的 EXC_RETURN 值来指定返回到线程模式时使用的栈。此外，在线程模式中，使用 MSR 指令对控制寄存器的第二位 CONTROL[1]执行写 1 操作也可以从主栈切换到进程栈。因此，栈指针 R13 是分组寄存器，在 SP_main 和 SP_process 之间切换。在任何时候，进程栈和主栈中只有一个是可见的，由 R13 指示。

栈指针寄存器 R13 中存放的是当前栈的地址，PUSH 和 POP 指令会自动对 R13 进行操作，其最低两位 R13[1：0]被强制置零，即它保存的地址是 4 字节对齐的。

3. 链接寄存器 R14

R14 链接寄存器(也记作 LR)用来保存返回地址。当主函数调用一个子函数时，就将主函数的下一条指令的地址(即子程序完成后的返回地址)保存到 LR，当子函数调用结束时返回到该地址便可继续执行主调函数；在发生异常中断时，LR 也用于特殊用途。其他任何时候都可以将 R14 看作一个通用寄存器。

4. 程序计数器 R15

R15 程序计数器(也记作 PC)指向下一条将要被执行的指令地址，该寄存器的最低位始终为 0，因此，指令始终与字或半字边界对齐，即指令是 16 位或 32 位的。如果向 PC 中写入地址，就会引起一次程序的跳转。读取 PC 将返回当前指令地址+4 的值，例如：

```
0x1000: MOV R0, PC
```

则

```
R0 = 0x1004
```

【思考题：读取 PC 将返回当前指令地址+4 的值原因是什么?**】**

5. 特殊功能寄存器组

Cortex-M3 的特殊功能寄存器包括：程序状态寄存器组 xPSR、中断屏蔽寄存器组(PRIMASK、FAULTMASK 以及 BASEPRI)和控制寄存器(CONTROL)，这些寄存器只能通过状态寄存器操作指令 MSR 和 MRS 在特权级别下访问。

(1) xPSR 程序状态寄存器由 3 个独立的状态寄存器组合而成，分别为应用程序状态寄存器(Application Program Status Register，APSR)、中断程序状态寄存器(Interrupt Program Status Register，IPSR)和执行程序状态寄存器(Execution Program Status

Register,EPSR)。其中,27～31 位为 APSR 所用,9～26 位为 EPSR 所用,0～8 位为 IPSR 所用。

APSR 是应用程序状态寄存器,用来保存条件代码标志。APSR 寄存器的有效域定义如图 2-5 所示。

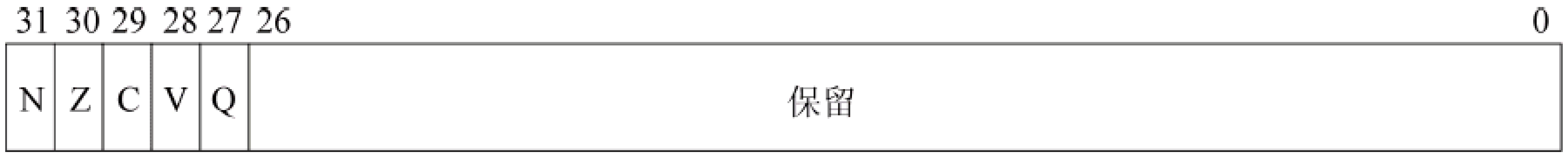


图 2-5　APSR 寄存器定义

N:负数或小于标志,1 表示结果为负数或小于,0 表示结果为整数或大于。

Z:零表示,1 表示结果为 0,0 表示结果非 0。

C:进位标志,1 表示有进位,0 表示没有进位。

V:溢出标志,1 表示有溢出,0 表示没有溢出。

Q:黏着饱和(sticky saturation)标志。

(2) 中断状态寄存器 IPSR,用于存放当前激活的异常的 ISR 编号。IPSR 的有效域定义如图 2-6 所示。

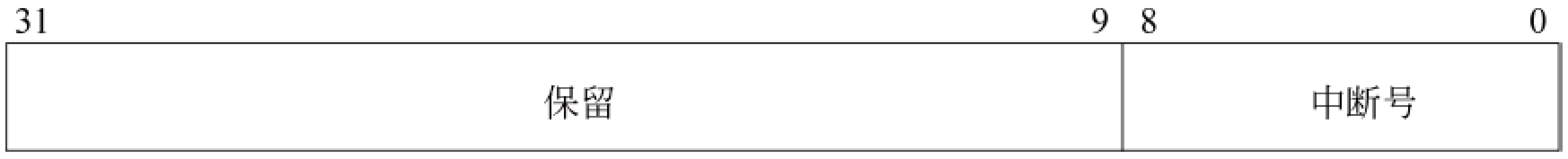


图 2-6　IPSR 寄存器定义

该寄存器的 0～8 位有效,用于表示中断服务编号,该域的值为 0 表示无中断,2 表示非屏蔽中断 NMI,16～255 表示 240 个外部中断。

(3) 执行状态寄存器 EPSR,该寄存器的 24～26 位,10～15 位有效,用于保存多寄存器连续加载过程中产生中断的相关信息。

3 个寄存器分别使用 32 位的不同区域,3 个寄存器可以单独访问,也可以 2 个或 3 个一起访问,在处理器进入异常时,处理器将 3 个状态寄存器组合的信息压入栈进行保存。

(4) 中断优先权屏蔽寄存器 PRIMASK 只有一个有效位,最低位为 1 时,关闭所有可屏蔽中断,只响应不可屏蔽中断 NMI 和硬件错误异常。

(5) 异常屏蔽寄存器 FAULTMASK 只有一个有效位,最低位为 1 时,关闭所有除不可屏蔽中断 NMI 外的所有异常。

(6) 中断优先级基准寄存器 BASEPRI 用来定义优先级的阈值,所有优先级大于该值的中断被关闭(优先级号越大,优先级越低)。

(7) 控制寄存器 CONTROL 只有两位有效,分别用于控制所使用的栈指针以及处理器工作在线程模式时的特权等级。CONTROL 的 bit[0]为 0 表示程序执行在特权级,bit[0]为 1 表示程序执行在非特权级。CONTROL 的 bit[1]为 0,表示选择主栈指针(MSP),bit[1]为 1,表示选择进程栈指针(PSP)。在异常响应模式下,只允许使用 MSP,所以此时不得往该位写 1。

2.2.4 总线

如图 2-7 所示，CM3Core 通过总线矩阵与代码存储器、数据存储器、外设和芯片内部外设的互联。ARM 处理器使用的总线规范是 AMBA，AMBA 规范主要包括 AHB 和 APB 两套总线。

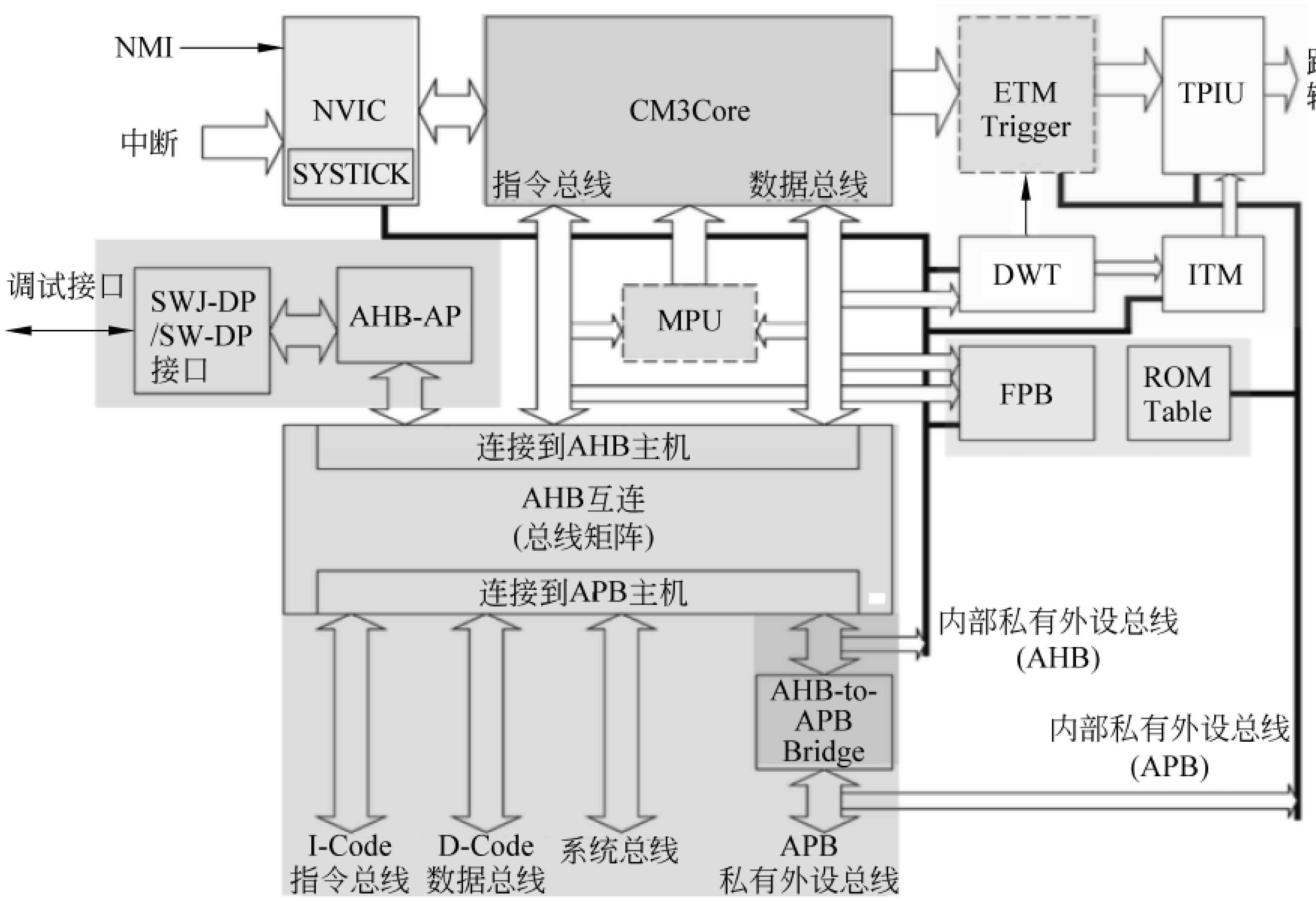


图 2-7 Cortex-M3 内部总线连接

AHB(Advanced High Performance Bus)用于高性能模块(如 CPU、DMA 和 DSP 等)之间的连接，其主要特性有：支持突发传输、分段传输，支持多个主控制器，可配置 32～128 位总线宽度等。AHB 是一套主从控制的总线，整个 AHB 总线上的传输都由主模块发出，由从模块负责回应，多个主从设备之间的数据通信由总线仲裁器进行管理。

AHB-Lite 是 AHB 总线的一个简化版本，只支持一个主模块，且没有总线仲裁器，简化了总线的复杂度。

APB(Advanced Pripheal Bus)总线主要用于低带宽的周边外设之间的连接，例如串口、USB 等，总线控制逻辑简单，只支持一个主模块 AHB/APB 桥接器，AHB 和 APB 通过桥接器进行数据连接和信号转换。

I-Code 总线和 D-Code 总线是 Cortex-M3 哈佛结构取指令和存取数据的两条基于 AHB-Lite 协议的 32 位总线，两条总线的地址访问范围均为 0x00000000～0x1FFFFFFF。I-Code 是指令总线，对于 16 位 Thumb 指令，一次可取两条指令；D-Code 是 32 位数据总线。

系统总线也是一条基于 AHB-Lite 总线协议的 32 位总线，负责在 0x20000000～0xDFFF_FFFF 和 0xE0100000～0xFFFFFFFF 两段地址空间的所有数据传送。

外部私有外设总线 PPB 是一条基于 APB 总线协议的 32 位总线。此总线用于 TPIU、ETM 以及 ROM 表等调试接口及外设。

一个典型的 Cortex-M3 和外部设备的总线连接实例如图 2-8 所示。

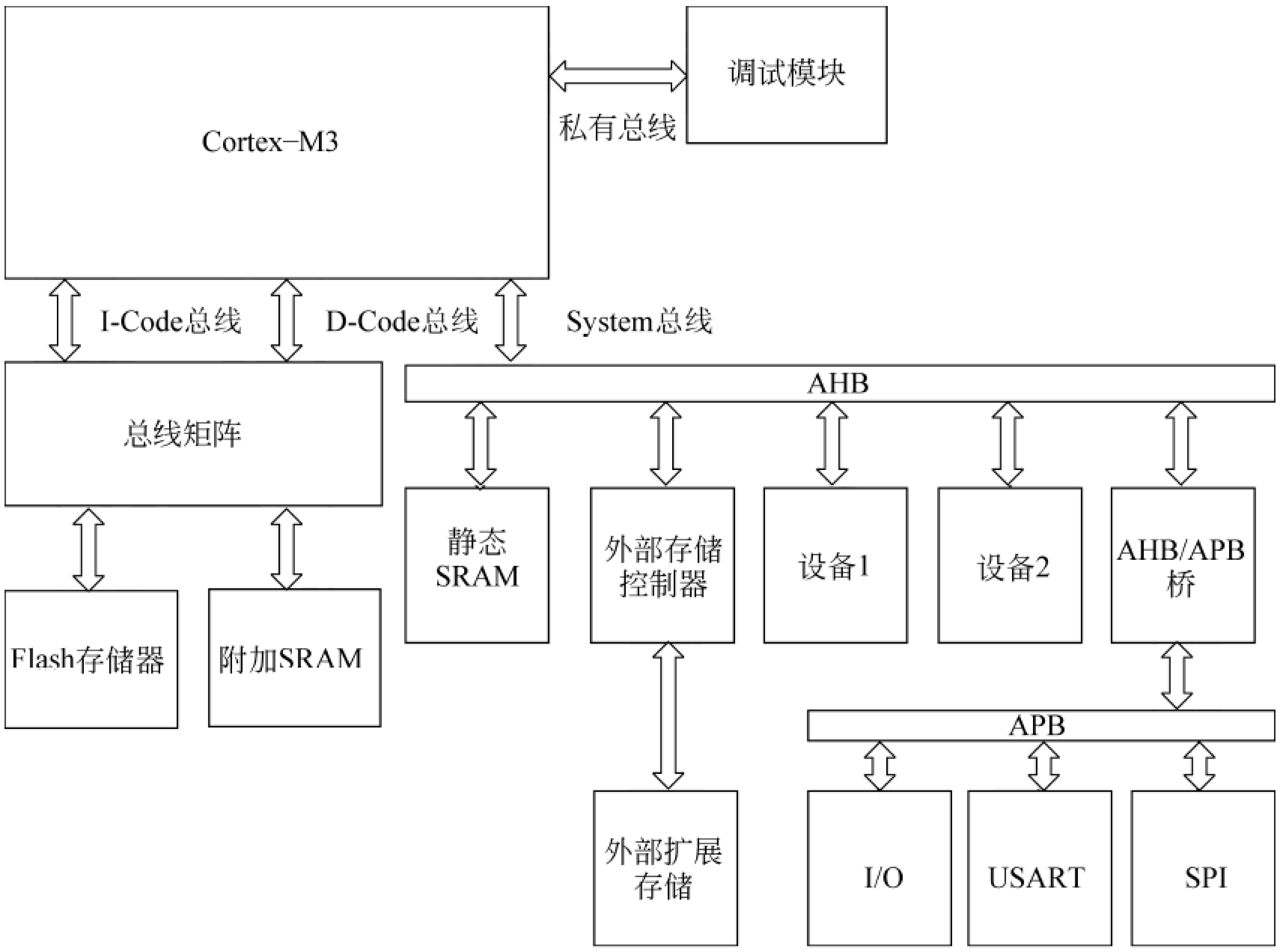


图 2-8 典型的总线连接结构

代码存储器既可以由指令总线(I-Code)访问,也可以被数据总线(D-Code)访问,总线矩阵可以实现指令和数据总线的分离。通过 AHB 总线矩阵把取指和数据访问分开后,如果指令总线和数据总线在同一时刻访问不同的存储器设备(例如,从 Flash 中取指的同时从附加的 SRAM 中访问数据)。但在一些系统实现时,没有附加 SRAM 或者使用了简化的总线复用器,则数据和指令无法同时进行传输。但微控制器内部集成的静态 RAM 一般连接在系统总线上,此时可以用 I-Code 访问 Flash 存储器、用 D-Code 访问系统总线 AHB 的 SRAM,实现哈佛结构。

为了增加系统的存储空间,可以在 AHB 总线上连接一个外部存储控制器,对外提供外部总线连接接口,实现 RAM 或 Flash 的扩容。

【思考题: 为何代码存储器既可以由指令总线访问,也可以被数据总线访问?**】**

2.2.5 存储器

1. 数据对齐

Cortex-M3 支持 32 位的字、16 位半字和 8 位的字节操作。通常情况下,要求总线上的数据要对齐,即以字为单位进行数据传输,其地址的最低两位必须是 0(地址是 4 的整数

倍);以半字为单位的传送,其地址的最低位必须是 0(地址是 2 的倍数)。如果使用了奇数地址,在一些处理器如 ARM7TDMI 中,会产生异常。Cortex-M3 支持非对齐的数据传输,其内部实际是通过把非对齐的访问转换成多个若干对齐的访问实现的,如图 2-9 的非对齐存储,需要通过两次存储器操作才能拿到数据,转换由总线单元完成,对程序员透明,但不是所有的地址空间都可以非对齐访问。一般情况下,我们采用对齐访问的方式,提高 CPU 的执行效率。

	byte3	byte2	byte1	byte0
Address *N*+4				[31:24]
Address *N*	[23:16]	[15:8]	[7:0]	

图 2-9 非对齐数据访问

2. 存储器格式

存储器是一个以字节为单位的线性存储空间,其地址可以从 0 开始向上编号,例如,字节 0～3 存放第一个被保存的字,字节 4～7 存放第二个被保存的字。但在存储半字和字的时候,一个半字或字的高字节和低字节的排列顺序可以不同,即存储器的大端和小端存储模式。

(1) 在小端格式中,一个字中最低地址的字节为该字的最低有效字节,最高地址的字节为最高有效字节,如图 2-10 所示。存储器系统地址 0 的字节与数据线 0～7 相连。

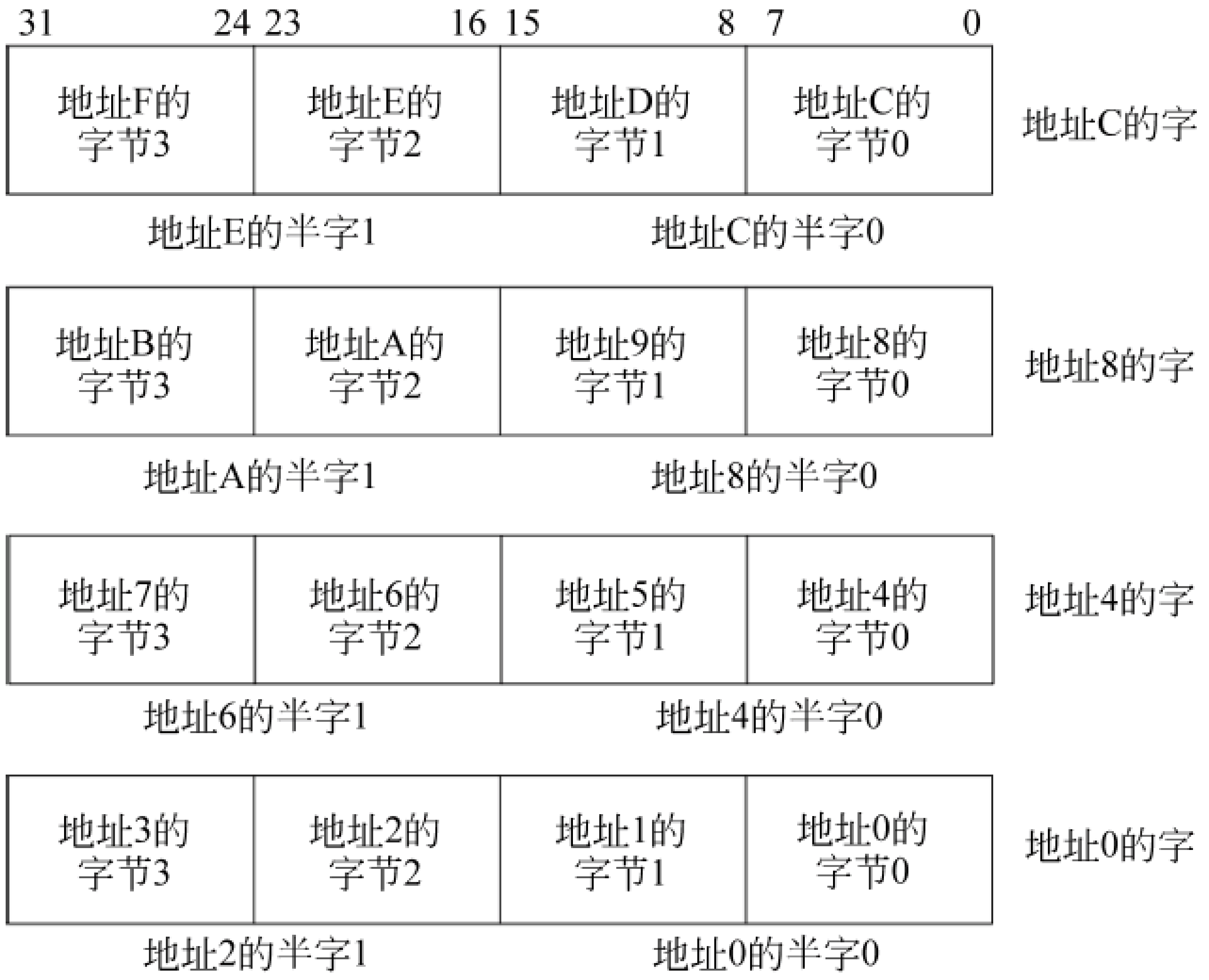


图 2-10 小端存储模式

(2) 在大端格式中,一个字中最低地址的字节为该字的最高有效字节,而最高地址的字节为最低有效字节,如图 2-11 所示。存储器系统地址 0 的字节与数据线 24～31 相连。

Cortex-M3 处理器有一个配置引脚 BIGEND,可以使用它来选择小端格式或大端格式。小端格式是 ARM 处理器默认的存储器格式。Cortex-M3 处理器能够以小端格式或大端格式访

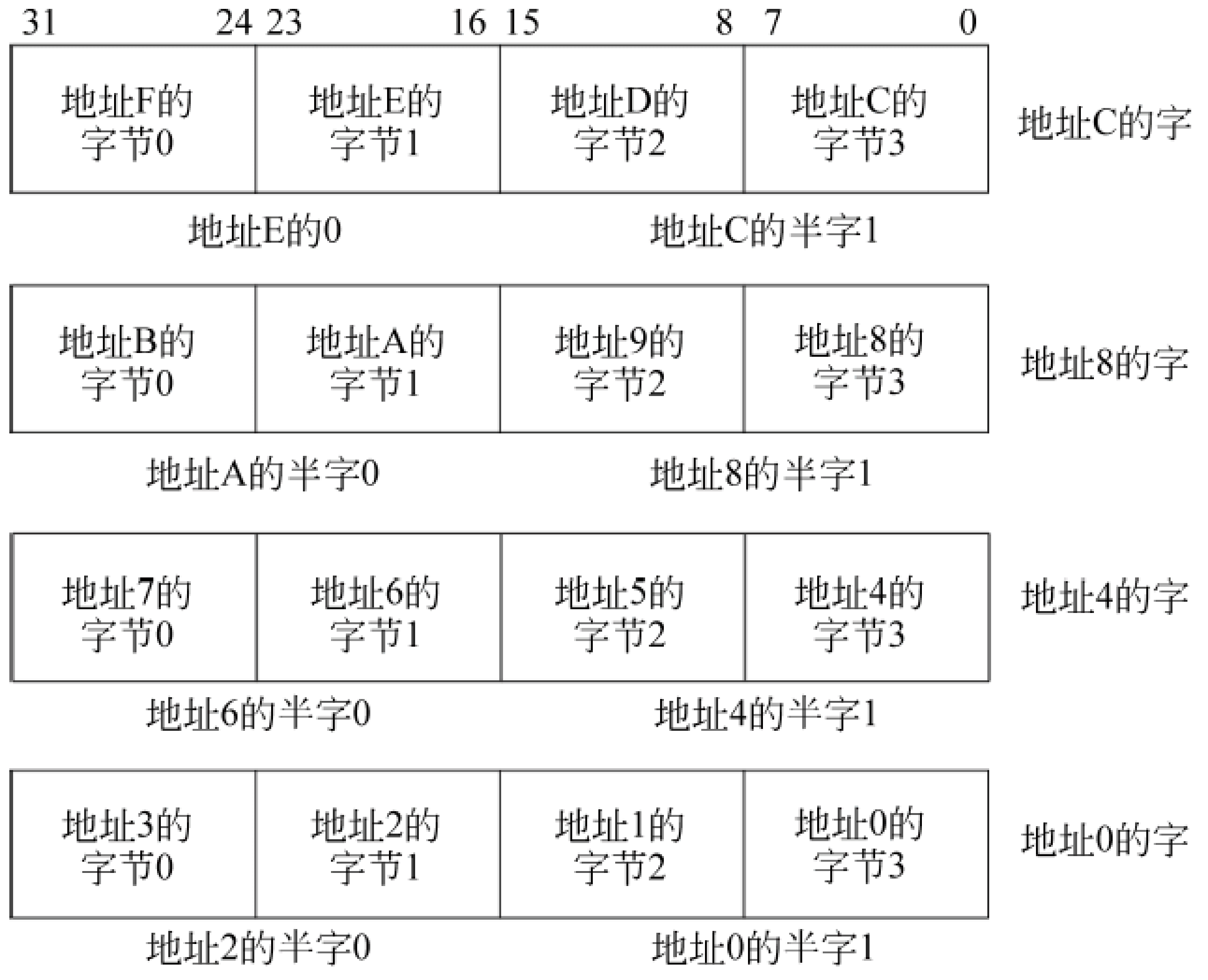


图 2-11　大端存储模式

问存储器中的数据字，而访问代码、系统控制空间 SCS 以及私有外设总线 PPB 空间时必须使用小端格式。

3. 存储器空间分配

Cortex-M3 采用统一编址，并对存储器空间分配进行了规范，Flash、SRAM 等起始地址以及 NVIC、MPU 的外设地址规定了具体的起始范围，这样使得不同厂家微控制器芯片的存储映射大体相同，便于在不同厂家微控制器之间的程序移植。

图 2-12 为 Cortex-M3 的存储器映射。

表 2-4 列出了被不同的存储器映射区域寻址的处理器接口。

表 2-4　存储器接口

存储器映射	接　　口
代码	指令取指在 I-Code 总线上执行，数据访问在 D-Code 总线上执行
SRAM	指令取指和数据访问都在系统总线上执行
SRAM_bitband	SRAM 的别名区域，数据访问是别名，指令访问不是别名
外设	指令取指和数据访问都在系统总线上执行
外设_bitband	外设别名区域，数据访问是别名，指令访问不是别名
外部 RAM	指令取指和数据访问都在系统总线上执行
外部设备	指令取指和数据访问都在系统总线上执行
专用外设总线	对 ITM、NVIC、FPB、DWT、MPU 的访问在处理器内部专用外设总线上执行。对 TPIU、ETM 和 PPB 存储器映射的系统区域的访问在外部专用外设总线上执行
系统	厂商系统外设的系统部分

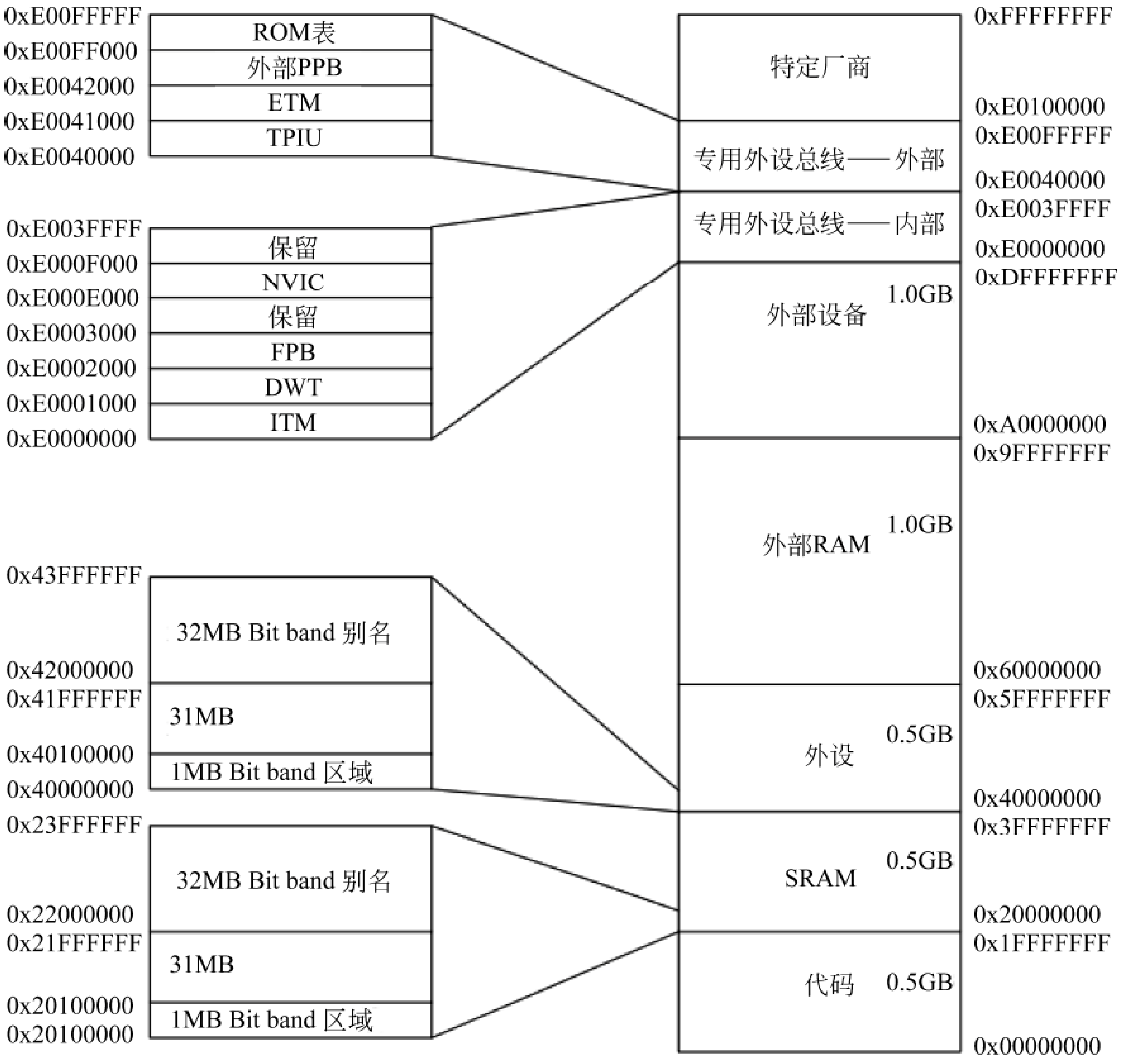


图 2-12　Cortex-M3 存储器映射

4. bit-banding 位带操作

bit-banding 技术是一种实现数据直接进行位操作的加速技术。如图 2-13 所示，存储器

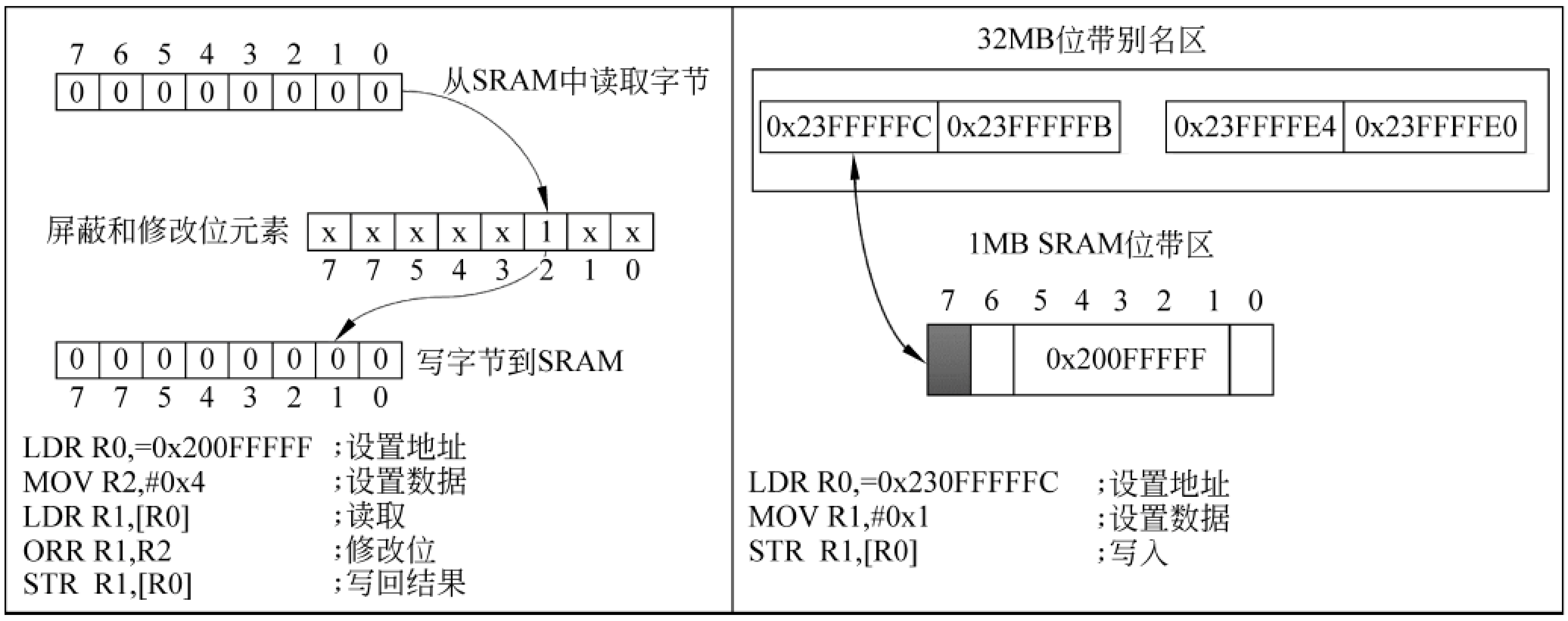


图 2-13　bit-banding 的对比

的最小访问单位是字节，通常情况下，要对一个字节中的某个 bit 进行操作，需要先读出该字节，对其中的某些位进行修改后再将整个字节写入到存储器中。Cortex-M3 的位带是在一段存储空间 A 中，将一个字的每一位(bit)映射到另一个存储空间 B 中的一个字(32bit)，对于 B 中字的操作即是对 A 中字的某一位的操作，从而通过一次读写即可实现对字节内部 bit 的直接操作。位带技术对于外设控制器的寄存器控制是非常有用的。我们把 B 称为位带别名区域。

Cortex-M3 存储器映射有 2 个 32MB 别名区，它们分别对应两个 1MB 的 bit-band 区。对 32MB SRAM 别名区的访问映射为对 1MB SRAM bit-band 区的访问。对 32MB 外设别名区的访问映射为对 1MB 外设 bit-band 区的访问。别名区中的字与 bit-band 区中对应的位的映射公式如下：

```
bit_word_offset= (byte_offset× 32)+ (bit_number× 4)
bit_word_addr=bit_band_base+bit_word_offset
```

其中，

- bit_word_offset 为 bit-band 存储区中的目标位的位置；
- bit_word_addr 为别名存储区中映射为目标位的字的地址；
- bit_band_base 是别名区的开始地址；
- byte_offset 为 bit-band 区中包含目标位的字节的编号；
- bit_number 为目标位的位位置(0～7)。

图 2-14 显示了 SRAM 位带别名区和 SRAM 位带区之间映射的一个例子：

别名区地址 0x23FFFFE0 的字映射为 0x200FFFFC 的 bit-band 字节的位 0：0x23FFFFE0＝0x22000000＋(0xFFFFF＊32)＋0＊4；别名区地址 0x23FFFFEC 的字映射为 0x200FFFFC 的 bit-band 字节的位 7：0x23FFFFFC＝0x22000000＋(0xFFFFF＊32)＋7＊4。

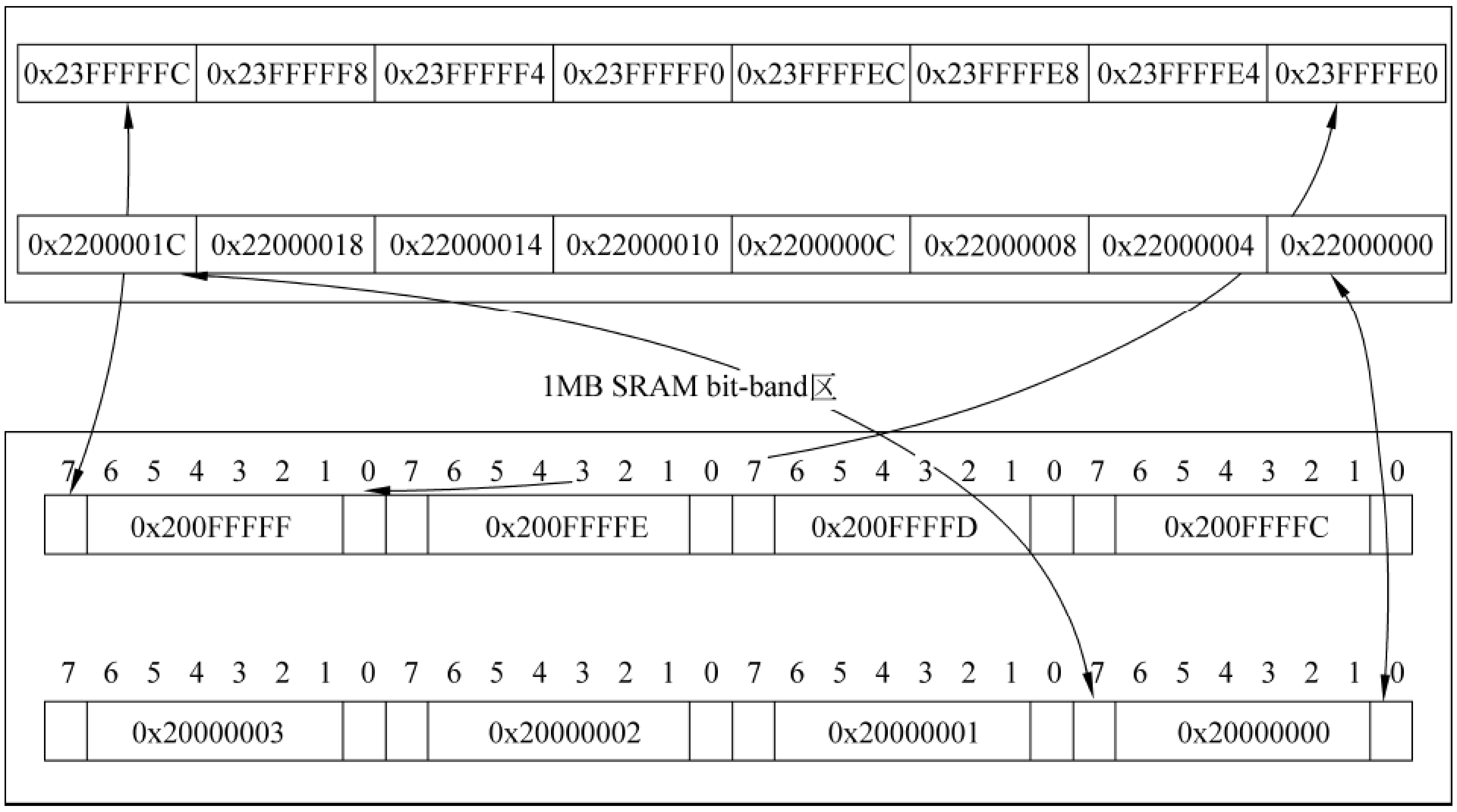


图 2-14　位带映射示例

2.2.6 中断

Cortex-M3 处理器使用一个可以重定位的向量表，表中包含了将要执行的中断服务程序的函数地址。中断被响应后，处理器通过指令总线接口从向量表中获取地址。向量表复位时存储地址为零，通过配置特殊寄存器可以使向量表重新定位。

当异常发生时，程序计数器、状态寄存器、链接寄存器和 R0～R3、R12 等通用寄存器将被压进栈。在数据总线对寄存器压栈的同时，处理器识别并定位异常向量，获取中断服务程序代码的第一条指令。一旦压栈和取指完成，中断服务程序就开始执行，中断服务程序执行完成，压栈的寄存器自动出栈恢复，中断了的程序也因此恢复正常的执行。

NVIC 支持中断嵌套(通过压栈实现)，允许通过提高中断的优先级对中断进行提前处理。它还支持中断的动态优先权重置。优先权级别可以在运行期间通过软件进行修改。在两个同级别中断发生的情况中，传统的系统将重复状态保存和状态恢复的过程两次，导致了延迟的增加。Cortex-M3 处理器使用末尾连锁(tail-chaining)技术简化了正在实行和将要执行的中断之间的移动。末尾连锁技术把需要用时 30 个时钟周期才能完成的连续的栈弹出和压入操作替换为 6 个周期就能完成的指令取指，实现了延迟的降低。处理器状态在进入中断时自动保存，在中断退出时自动恢复，比软件执行用时更少，大大提高了系统的性能，NVIC 中断控制器的细节将在第 6 章中断中进行详细介绍。

2.3 STM32L152RET6 微处理器介绍

本教材选用了意法半导体公司的 STM32L 系列超低功耗处理器 STM32L152RET6，该处理器属于 Cortex-M3 系列。

对于超低功耗处理器 STM32L152RET6，其涉及的命名规则如下：

(1) STM32 表示基于 ARM 的 32 位微处理器。

(2) L 表示低功耗产品系列，F 表示通用产品系列。

(3) 152 是产品系列中子类型产品。

后续的四位编号 RET6 的说明如下：

(4) 第一位表示引脚数量：R 表示 64 引脚，T 表示 36 引脚，C 表示 46 引脚，V 表示 100 引脚，Z 表示 144 引脚。

(5) 第二位表示 Flash 大小：B 表示 Flash 容量为 128KB，6 表示 Flash 容量为 32KB，8 表示 Flash 存储容量为 64KB，C 表示 Flash 存储容量为 256KB，D Flash 存储容量 384KB，E 表示 Flash 存储容量为 512KB，G 表示 Flash 存储容量为 1MB。

(6) 第三位表示封装方式：T 表示 LQFP 封装，H 表示 BGA 封装，U 表示 VFQFPN 封装。

(7) 第四位表示工作温度：6 表示工业级，工作温度为－40℃～85℃；当为 7 时，表示工

作温度为－40℃～105℃。

STM32L152RET6 主要配置包括：512KB 的 Flash、80KB 的 RAM、16KB EEPROM、2 个超低功耗比较器、6 个通用计时器和 2 个基本计时器、2 个 SPI 通信接口、2 个 I2C 通信接口、3 个 USART 通信接口和 1 个 USB 接口、51 个常规输入输出端口(分为 6 组)、1 个 12 位 20 通道的模数转换器、2 个 12 位 2 通道的数模转换器、主频最大为 32MHz、工作电压在 1.8～3.6V 之间。

当然，作为超低功耗处理器 STM32L152RET6，其支持 7 种类低功耗工作模式，待机模式最低功耗可达 290nA，唤醒时间 8uS：

(1) 睡眠模式：只有 CPU 停止工作，所有其他外部设备继续运行，当中断或事件发生时可以唤醒 CPU。

(2) 低功耗运行模式：内部时钟工作频率为 65kHz，外部使能设备数量受限。

(3) 低功耗睡眠模式：在睡眠模式下通过将内部电压调节器设置为低功耗模式以减少电压调节器的工作电流，外部使能设备数量受限。

(4) 带 RTC 的停止模式：同时保持 RAM 和寄存器的内容以及实时时钟，低速时钟工作，电压调节器工作在低功耗模式可以由外部中断唤醒。

(5) 不带 RTC 的停止模式：保持 RAM 和寄存器的内容，所有时钟均停止工作，电压调节器工作在低功耗模式，可以由外部中断唤醒。

(6) 带 RTC 的待机模式：内部电压调节器被关闭。低速时钟仍然运行，RAM 和大多数寄存器的内容丢失；待机模式可由复位(NRST 引脚信号、独立看门狗)、唤醒引脚以及 RTC 事件触发退出。

(7) 不带 RTC 的待机模式：内部电压调节器被关闭，所有时钟均停止工作，RAM 和大多数寄存器的内容会丢失。这种待机模式可由外部复位(NRST 引脚信号)或唤醒引脚来触发退出。

2.4　STM32L152RET6 微处理器的系统结构

STM32L152RET6 微处理器的系统结构，如图 2-15 所示。

I-Code 总线将 Cortex-M3 内核与闪存(Flash)指令接口相连接，指令通过该总线传输。D-Code 总线将 Cortex-M3 内核与 Flash 数据接口相连接。System 总线连接 Cortex-M3 内核的 System 总线到总线矩阵，总线矩阵协调 Cortex-M3 内核和 DMA 的访问。DMA 总线将连接 DMA 到总线矩阵。

总线矩阵协调 Cortex-M3 的 System 总线和 DMA 总线之间的访问仲裁。仲裁采用轮换算法。总线矩阵包含五个主动部件，包括 Cortex-M3 的 D-Code 总线、Cortex-M3 的 System 总线、以太网 DMA 总线、DMA1 总线和 DMA2 总线，三个从属部件，包括 Flash 接口、SRAM 和 AHB/APB 桥。

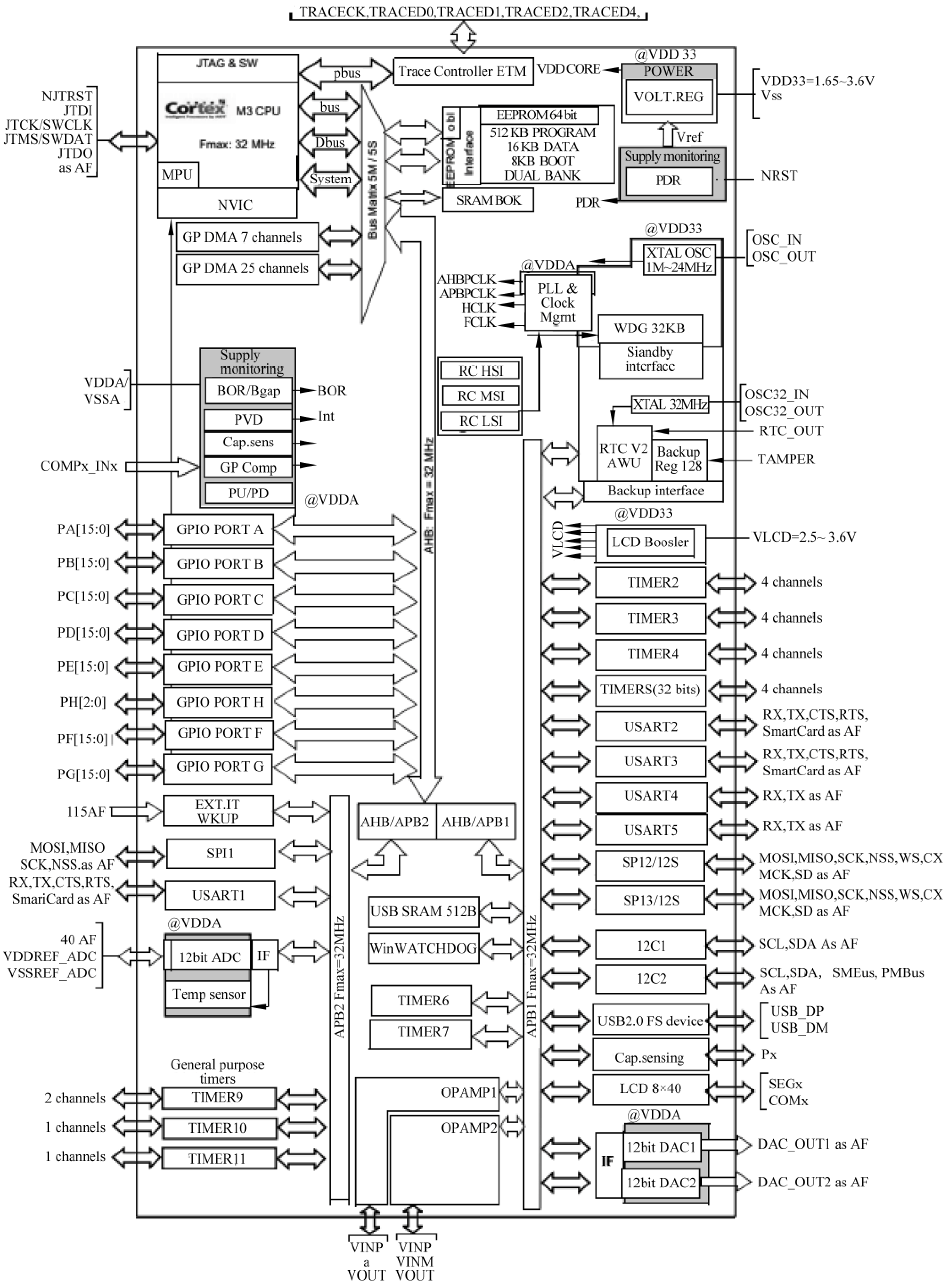


图 2-15　STM32L152RET6 的系统结构

两个 AHB/APB 桥在 AHB 和 2 个 APB 总线间提供同步连接。当 APB 进行 8 位或 16 位访问时，该访问会自动转换成 32 位的访问。

APB 总线上连接的外设主要有：数模转换器(Digital Analog Converter，DAC)、电源(Power，PWR)、USB(Universal Serial Bus)设备、I2C(Inter Integrate Circuit)总线、USART(Universal Serial Asynchronous Receiver Transmitter)总线、SPI(Serial Peripheral Interface)总线、IWDG(Independent Watchdog)独立看门狗、WWDG(Window Watchdog)窗口看门狗、RTC(Real Time Clock)实时时钟、实时器(Timer)等。

I2C 总线是由 PHILIPS 公司开发的两线式串行总线，一条是 SDA(Serial Data)串行数据线，另一条是 SCL(Serial Clock)串行时钟线，用于连接微控制器及其外围设备。I2C 是微电子通信控制领域广泛采用的一种总线标准，是同步通信的一种特殊形式，具有接口线少，控制方式简单，器件封装形式小，通信速率较高等优点。

USART 是一种通用串行数据总线，用于异步通信。该总线双向通信，可以实现全双工传输和接收。在嵌入式设计中，USART 用来与 PC 进行通信，包括与监控调试器和其他器件，如 EEPROM 通信。USART 首先将接收到的并行数据转换成串行数据来传输。消息帧从一个低位起始位开始，后面是 7 个或 8 个数据位，一个可用的奇偶位和一个或几个高位停止位。接收器发现开始位时它就知道数据准备发送，并尝试与发送器时钟频率同步。如果选择了奇偶，UART 就在数据位后面加上奇偶位。奇偶位可用来帮助错误校验。接收过程中，UART 从消息帧中去掉起始位和结束位，对进来的字节进行奇偶校验，并将数据字节从串行转换成并行。UART 也产生额外的信号来指示发送和接收的状态。例如，如果产生一个奇偶错误，UART 就置位奇偶标志。

SPI 总线是一种高速的、全双工、同步的通信总线，并且在芯片的引脚上只占用四根线，节约了芯片的引脚，同时为 PCB 的布局上节省空间，提供方便。

独立看门狗 IWDG 是一个 12 位的向下计数器，其时钟来自于其内部独立的 37kHz 的 RC 振荡器(只用电阻和电容构成的振荡器称为 RC 振荡器)。该看门狗可用于当故障发生时重置设备，还可用于应用超时管理的自激时钟。独立看门狗没有中断。一般主要用于监视硬件错误。

窗口看门狗 WWDG 是一个 7 位的向下计数器，可用于当问题发生时重置设备。其时钟来源于主时钟。窗口看门狗有中断。一般主要用于监视软件错误。

STM32 的实时时钟(RTC)是一个独立的定时器。STM32 的 RTC 模块拥有一组连续计数的计数器，在相应软件配置下，可提供时钟日历的功能。修改计数器的值可以重新设置系统当前的时间和日期。RTC 模块和时钟配置系统(RCC_BDCR 寄存器)是在后备区域，即在系统复位或从待机模式唤醒后 RTC 的设置和时间维持不变。但是在系统复位后，会自动禁止访问后备寄存器和 RTC，以防止对后备区域(BKP)的意外写操作。所以在要设置时间之前，先要取消备份区域(BKP)写保护。

2.5　STM32L152RET6 微处理器的引脚说明

STM32L152RET6 微处理器的引脚，如图 2-16 所示。

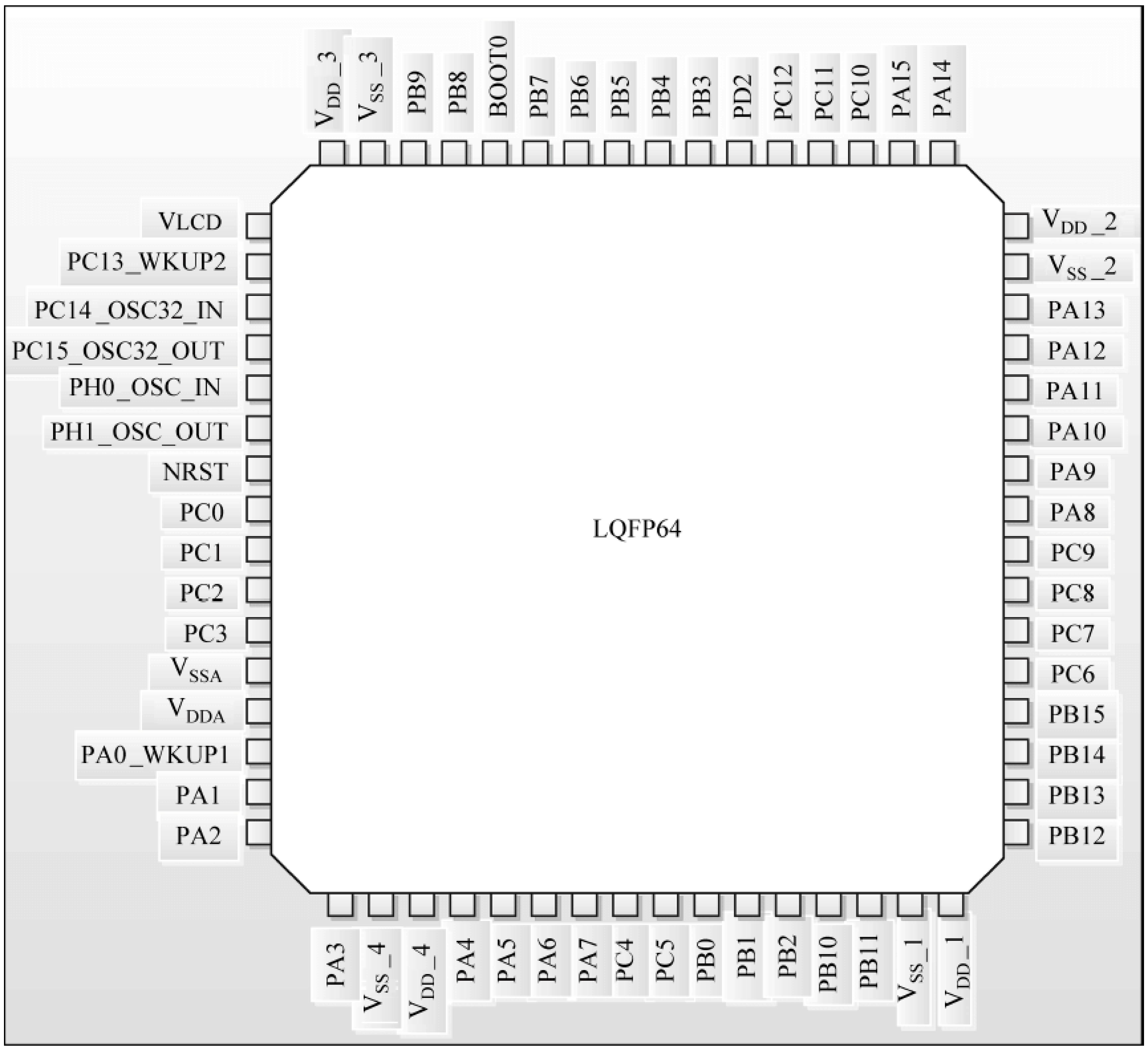


图 2-16　STM32L152RET6 的引脚

各个引脚的具体功能，如表 2-5 所示。其中，引脚类型为 S 表示供电管教，I/O 表示输入输出，I 表示只能输入。I/O 电平 FT 表示可支持 5V 电压，TC 表示支持 3V 电压。

表 2-5　STM32L152RET6 引脚的功能说明

编号	引脚名	引脚类型	I/O 电平	主要功能	附 加 功 能
1	V_{LCD}	S		V_{LCD}	
2	PC13_WKUP2	I/O	FT	PC13	RTC_TAMP1、RTC_TS、RTC_OUT、WKUP2
3	PC14_OSC32_IN	I/O	TC	PC14	OSC32_IN
4	PC15_OSC32_OUT	I/O	TC	PC15	OSC32_OUT

续表

编号	引脚名	引脚类型	I/O 电平	主要功能	附 加 功 能
5	PH0_OSC_IN	I/O	TC	PH0	OSC_IN
6	PH1_OSC_OUT	I/O	TC	PH1	OSC_OUT
7	NRST	I/O	RST	NRST	
8	PC0	I/O	FT	PC0	ADC_IN10、COMP1_INP
9	PC1	I/O	FT	PC1	ADC_IN11、COMP1_INP
10	PC2	I/O	FT	PC2	ADC_IN12、COMP1_INP
11	PC3	I/O	TC	PC3	ADC_IN13、COMP1_INP
12	V_{SSA}	S		V_{SSA}	
13	V_{DDA}	S		V_{DDA}	
14	PA0_WKUP1	I/O	FT	PA0	WKUP1、ADC_IN0、COMP1_INP
15	PA1	I/O	FT	PA1	ADC_IN1、COMP1_INP
16	PA2	I/O	FT	PA2	ADC_IN2、COMP1_INP
17	PA3	I/O	FT	PA3	ADC_IN3、COMP1_INP
18	V_{SS_4}	S		V_{SS_4}	
19	V_{DD_4}	S		V_{DD_4}	
20	PA4	I/O	TC	PA4	ADC_IN4、DAC_OUT1、COMP1_INP
21	PA5	I/O	TC	PA5	ADC_IN5、DAC_OUT2、COMP1_INP
22	PA6	I/O	FT	PA6	ADC_IN6、COMP1_INP
23	PA7	I/O	FT	PA7	ADC_IN7、COMP1_INP
24	PC4	I/O	FT	PC4	ADC_IN14、COMP1_INP
25	PC5	I/O	FT	PC5	ADC_IN15、COMP1_INP
26	PB0	I/O	TC	PB0	ADC_IN8、COMP1_INP、VREF_OUT
27	PB1	I/O	FT	PB1	ADC_IN9、COMP1_INP、VREF_OUT
28	PB2	I/O	FT	PB2/BOOT1	
29	PB10	I/O	FT	PB10	
30	PB11	I/O	FT	PB11	
31	V_{SS_1}	S		V_{SS_1}	

续表

编号	引脚名	引脚类型	I/O 电平	主要功能	附 加 功 能
32	V_{DD_1}	S		V_{DD_1}	
33	PB12	I/O	FT	PB12	ADC_IN18、COMP1_INP
34	PB13	I/O	FT	PB13	ADC_IN19、COMP1_INP
35	PB14	I/O	FT	PB14	ADC_IN20、COMP1_INP
36	PB15	I/O	FT	PB15	ADC_IN21、COMP1_INP、RTC_REFIN
37	PC6	I/O	FT	PC6	
38	PC7	I/O	FT	PC7	
39	PC8	I/O	FT	PC8	
40	PC9	I/O	FT	PC9	
41	PA8	I/O	FT	PA8	
42	PA9	I/O	FT	PA9	
43	PA10	I/O	FT	PA10	
44	PA11	I/O	FT	PA11	USB_DM
45	PA12	I/O	FT	PA12	USB_DP
46	PA13	I/O	FT	JTMS_SWDIO	
47	V_{SS_2}	S		V_{SS_2}	
48	V_{DD_2}	S		V_{DD_2}	
49	PA14	I/O	FT	JTCK_SWCLK	
50	PA15	I/O	FT	JTD1	
51	PC10	I/O	FT	PC10	
52	PC11	I/O	FT	PC11	
53	PC12	I/O	FT	PC12	
54	PD2	I/O	FT	PD2	
55	PB3	I/O	FT	JTDO	
56	PB4	I/O	FT	NJRST	
57	PB5	I/O	FT	PB5	
58	PB6	I/O	FT	PB6	
59	PB7	I/O	FT	PB7	
60	BOOT0	I	B	BOOT0	

续表

编号	引脚名	引脚类型	I/O 电平	主要功能	附 加 功 能
61	PB8	I/O	FT	PB8	
62	PB9	I/O	FT	PB9	
63	V_{SS_3}	S		V_{SS_3}	
64	V_{DD_3}	S		V_{DD_3}	

2.6　STM32L152RET6 微处理器的复位和时钟控制

STM32L152RET6 微处理器支持三种形式的复位：电源复位、系统复位和备份区域复位。

电源复位是指复位所有寄存器。当发生系统上电、掉电或欠压复位时，发生电源复位，这些复位源都作用在 NRST 引脚，提供给设备的系统复位信号都由 NRST 引脚输出。

系统复位是指设置除时钟控制寄存器和备份区域寄存器外的所有寄存器。系统复位可由如下几种方式产生：①引脚 NRST 低电平；②窗口看门狗计数终止；③独立看门狗计数终止；④软件复位；⑤低功耗管理复位，即通过进入待机模式（Standby）或停止模式（STOP）产生的复位；⑥待机模式退出复位。

备份区域复位仅仅设置备份区域寄存器。有以下两种方式产生：①通过设置备份区域控制寄存器 RCC_BDCR 的 BDRST 位置为 1 产生的软件复位；②电源复位。

STM32L152RET6 微处理器共有 5 个时钟，3 个为内部时钟，分别为高速内部时钟（High Speed Internal clock signal，HSI）、低速内部时钟（Low Speed Internal clock signal，LSI）和多速内部时钟（Multi-Speed Internal clock signal，MSI）。两个为外部时钟，分别为高速外部时钟（High Speed External clock signal，HSE）和低速外部时钟（Low Speed External clock signal，LSE）。STM32L152RET6 微处理器的时钟树，如图 2-17 所示。其中，高速外部时钟 HSE 的频率一般在 1M～24MHz 范围内，低速外部时钟 LSE 的频率一般在 0～1000kHz 范围内，一般取典型值 32.768kHz。

【思考题：为何采用多时钟？】

三种不同的主时钟源可被用来驱动系统时钟（SYSCLK）：

（1）16MHz 的 HSI 内部高速振荡器时钟。

（2）1M～24MHz 的 HSE 外部高速振荡器时钟。

（3）MSI 多速率时钟（7 种可配置时钟速率，65.5kHz、131kHz、262kHz、524kHz、1.05MHz、2.1MHz 和 4.2MHz）。

HSE 和 HSI 可以作为 PLL 锁相环的输入源。PLL 是一个倍频模块，可以把低速率时钟倍频后输出高速率时钟，PLL 锁相环的输出时钟经过分频后可以作为 SYSCLK 的时钟源。

【**思考题**：什么是 PLL？它的原理是什么？】

两个辅助时钟源可用于驱动 LCD 控制器以及 RTC 等：

(1) 32.768kHz 的 LSE 时钟。

(2) 37kHz 的 LSI 时钟。

当不使用时，任一个时钟源都可被独立地启动或关闭，由此优化系统功耗。

图 2-17 是 STM32L1xx 系列处理器的时钟树，用户可通过多个预分频器配置 AHB、高速 APB（APB2）和低速 APB（APB1）域的频率。AHB、APB1 和 APB2 域的最大频率是 32MHz，具体取决于芯片的工作电压。所有的外围设备的时钟都由 SYSCLK 主时钟产生，除了以下几种情况：

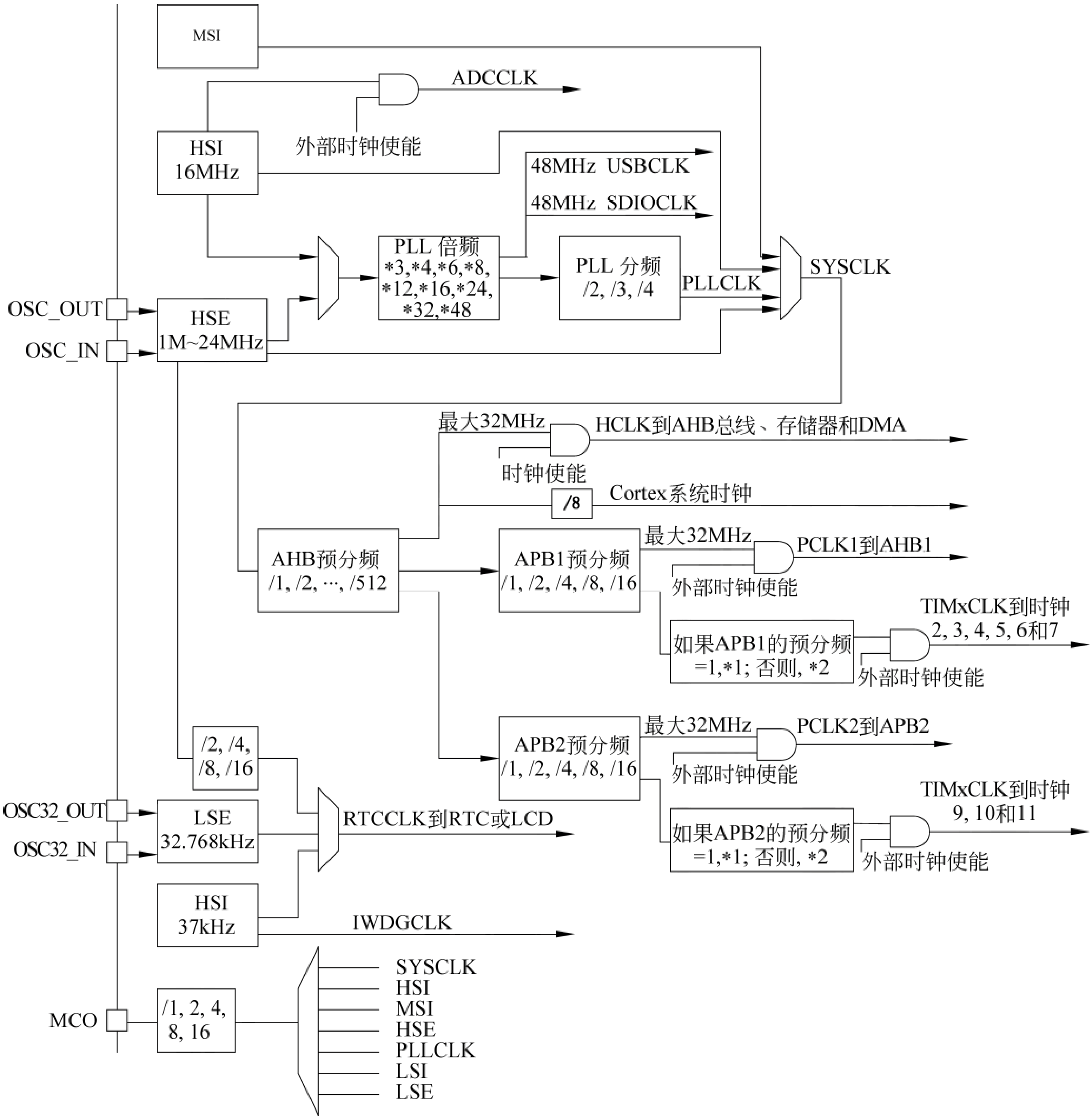


图 2-17　STM32L152RET6 微处理器的时钟树

(1) USB 和 SDIO 的 48MHz 时钟由 PLLVCO 产生。

(2) ADC 由 HIS 时钟 1,2,4 分频产生。

(3) IWDG 由 LSI 时钟提供。

(4) RTC 和 LCD 可以由 LSE、LSI 和 HSE 提供。

时钟由 RCC(Reset and Clock Contorller)控制器进行配置和管理,RCC 的详细介绍见第四章。

2.7 STM32L152RET6 微处理器的存储映射

STM32L152RET6 微处理器的存储映射,如图 2-18 所示。

程序存储器(Flash)、数据存储器(SRAM)、寄存器和输入输出端口(GPIO)统一组织在一个 4GB 的存储区域,即统一编址。如果要访问输入输出端口,就向对应的地址写入数据;如果要设置输入输出端口的属性,就要写信息到相应的寄存器。

代码区始终从地址 0x00000000 开始,且通过 I-Code 和 D-Code 总线访问,而数据区始终从地址 0x2000 0000 开始,且通过系统总线访问。

从地址 0x00000000 到地址 0x1FFFFFFF 共 512MB 空间为块 0,根据 BOOT[1:0]引脚的设置从主闪存存储器(Flash Memory)、系统存储器(System Memory)或内置 SRAM 启动。即 BOOT 的设置将 Flash、System Memory 和 SRAM 映射到 0x00000000 开始的空间。

通过 BOOT[1:0]引脚的设置可以选择三种不同的启动模式,如表 2-6 所示。

表 2-6 STM32L152RET6 的启动模式

BOOT[1:0]引脚设置		启动模式	说明
BOOT1	BOOT0		
x	0	主闪存存储器(Flash)	主闪存被选为启动区域
0	1	系统存储器(System Memory)	系统存储器被选为启动区域
1	1	内置 SRAM	内置 SRAM 被选为启动区域

对于不同的启动模式,开始访问的地址空间不同。

(1) 从主闪存存储器(Flash Memory)启动。主闪存存储器被映射到启动空间地址 0x0000 0000~0x07FF FFFF 中,但仍然能够访问原有的从地址 0x0800 0000 开始的空间;

(2) 从系统存储器(System Memory)启动。系统存储器被映射到启动空间地址 0x0000 0000~0x07FF FFFF 中,但仍然能够访问原有的从地址 0x1FF0 0000 开始的空间;

(3) 从内置 SRAM 启动。只能在地址 0x2000 0000 开始的空间访问 SRAM。

从地址 0x2000 0000 到地址 0x3FFF FFFF 共 512MB 空间为块 1,为 SRAM 区。

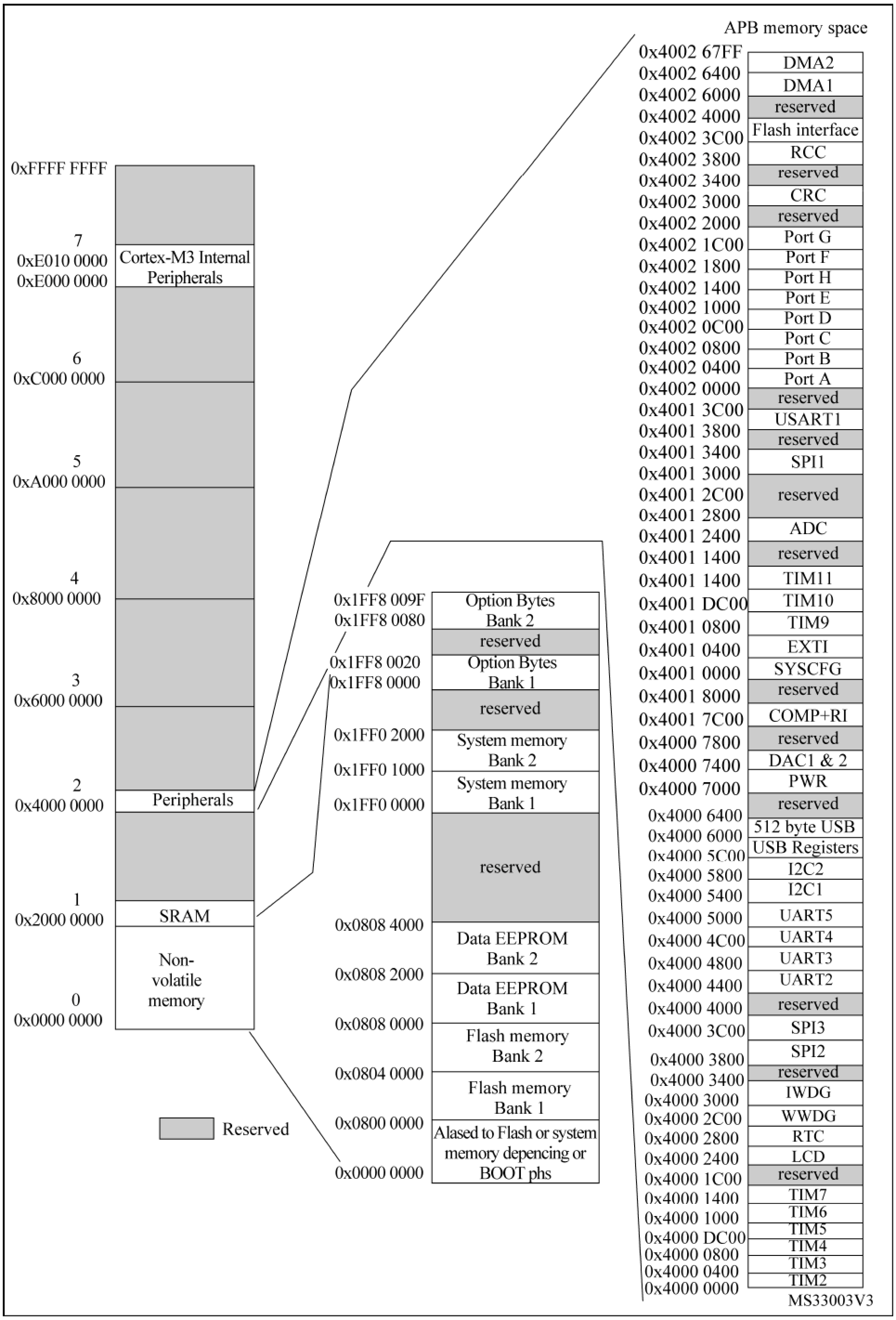


图 2-18　STM32L152RET6 的存储器映射

从地址 0x4000 0000 到地址 0x5FFF FFFF 共 512MB 空间为块 2,为外部设备区,相关的寄存器组及地址,如表 2-7 所示。

表 2-7　寄存器组地址

地　　址	外　　设	总线
0x40000000～0x400003FF	TIM2 定时器	APB1
0x40000400～0x400007FF	TIM3 定时器	APB1
0x40000800～0x40000BFF	TIM4 定时器	APB1
0x40001000～0x400013FF	TIM6 定时器	APB1
0x40001400～0x40001BFF	TIM7 定时器	APB1
0x40002400～0x400027FF	LCD	APB1
0x40002B00～0x40002BFF	RTC	APB1
0x40002C00～0x40002FFF	WWDG 窗口看门狗	APB1
0x40003000～0x400033FF	IWDG 独立看门狗	APB1
0x40003800～0x40003BFF	SPI2	APB1
0x40004400～0x400047FF	USART2	APB1
0x40004800～0x40004BFF	USART3	APB1
0x40005400～0x400057FF	I2C1	APB1
0x40005800～0x40005BFF	I2C2	APB1
0x40005C00～0x40005FFF	USB 寄存器	APB1
0x40006000～0x400061FF	USB 512B	APB1
0x40007000～0x400073FF	PWR 电源控制	APB1
0x40007400～0x400077FF	DAC	APB1
Ox40007C00～0x40007EFF	COMP＋RI	AHB
0x40010400～0x400107FF	EXTI	APB2
0x40010800～0x40010BFF	TIM9	APB2
0x40010C00～0x40010FFF	TIM10	APB2
0x40011000～0x400113FF	TIM11	APB2
0x40012400～0x400127FF	ADC	APB2
0x40013000～0x400133FF	SPI1	APB2
0x40013800～0x40013BFF	USART1	APB2
0x40020000～0x400203FF	GPIO 端口 A	AHB
0x40020400～0x400207FF	GPIO 端口 B	AHB

续表

地　　址	外　　设	总线
0x40020800～0x40020BFF	GPIO 端口 C	AHB
0x40020C00～0x40020FFF	GPIO 端口 D	AHB
0x40021400～0x400217FF	GPIO 端口 H	AHB
0x40023000～0x400233FF	CRC	AHB
0x40023800～0x40023BFF	RCC 复位和始终控制	AHB
0x40023C00～0x40023FFF	闪存存储器接口	AHB
0x40026000～0x400263FF	DMA	AHB

Chapter 3

第 3 章 Cortex-M3 处理器的指令系统

【导读】 Cortex-M3 处理器支持的指令集包括 ARMv6 的大部分 16 位 Thumb 指令和 ARMv7 的 Thumb-2 指令集。本章给出了指令的一般格式，并详细说明了存储器访问指令、数据处理指令、乘法除法指令、位操作指令和分支控制指令等使用，最后在说明符号定义伪指令、数据定义伪指令、汇编控制伪指令和宏指令的使用方法后，举例说明了汇编程序的编写方法。目前嵌入式开发主要以高级语言为主，汇编语言作为性能调优和底层初始化使用，本章介绍的指令集旨在让读者了解指令的种类、功能和基本使用方法，读懂汇编程序代码。

3.1 Cortex-M3 处理器的指令系统

3.1.1 指令系统基本概念

1. 指令和指令集

冯·诺依曼体系结构采用存储程序的原则，即事先将程序存储到计算机中，程序是由计算机可执行的指令组成的，计算机的控制器根据程序指令控制计算机自动运行，即计算机执行程序的过程就是执行一条条指令的过程。

指令是指 CPU 能直接识别并执行的控制命令，它的表现形式是二进制编码。指令通常由操作码和操作数两部分组成，操作码指出该指令所要完成的操作，即指令的功能，操作数指出参与运算的数据或其存储地址等。

【思考题：为何指令中要使用数据存储地址而非直接使用数据?**】**

由于指令与 CPU 紧密相关，不同类型的 CPU 所对应的指令也有所不同，一台计算机所能执行的各种不同指令的全体，称为计算机的指令系统。一般同一系列的 CPU 指令集具有兼容性，即新的指令系统必须包括先前同系列 CPU 的指令系统，这样先前开发的各类程序在新 CPU 上才能正常运行。

机器语言是用来直接描述指令的最佳语言，是 CPU 能直接识别的唯一一种计算机语言，但机器语言书写大规模程序不容易维护。汇编语言使用助记符来代替和表示特定机器语言操作，相对机器语言更容易理解和维护，且汇编语言中可使用标签和符号等代表操作数或功能模块，使得程序更加灵活。一般汇编语言和其特定的机器语言描述的指令集是一一

对应的，但 CPU 无法识别汇编语言，因此需要汇编器进行转换。汇编语言目前已不像十几年前被广泛用于程序设计，只在操作系统、驱动程序等底层硬件操作和极高要求的程序优化场合下使用。

指令按照不同的分类可划分不同的种类。按照指令实现的功能，指令可分为：数据传送指令、算术运算指令、逻辑运算指令、程序控制指令、系统控制指令等。数据传送指令用来实现主存与 CPU 寄存器以及寄存器与寄存器之间的数据传输，例如 Thumb-2 的取一个字到寄存器指令 LDR、寄存器数据存储到主存指令 STR，寄存器间数据传送指令 MOV 等。算术运算是计算机能够执行的基本数值计算，算术运算指令包括加法 ADD、减法 SUB、乘法 MUL、除法 DIV 等指令。逻辑运算是对数据进行逻辑操作，包括逻辑与 AND、逻辑或 ORR、逻辑非 MVN 等三种基本操作以及同或、异或等组合逻辑操作。程序控制指令主要包括：转移指令 BL、断点指令 BKRT、分支指令 IT 等。系统控制指令包括休眠指令 WFI、WFE、空操作指令 NOP、开关中断指令 CPSIE、CPSID 等。

按照数据存取方式，指令可分为寄存器-寄存器型、寄存器-存储器型和存储器-存储器型。寄存器-寄存器型将数据存放在寄存器中进行操作，例如 Cortex-M3 的大多数数据传输和算术逻辑运算指令，寄存器-寄存器型指令编码简单、执行速度快，指令周期相近。寄存器-存储器型指令可直接对存储器操作数进行访问，如访存操作 LDR、STR 等，指令周期相差较大，执行速度较慢。存储器-存储器型无需寄存器保存数据，其执行的访存较多，执行速度慢，Cortex-M3 绝大多数指令不采用存储器-存储器结构。

2. Cortex-M3 指令的编码方式

Cortex-M3 支持 16 位和 32 位的 Thumb-2 指令集，一个典型的 16 位指令的编码如图 3-1 所示。

15 14 13 12 11 10	9 8 7 6 5 4 3 2 1 0
opcode	

图 3-1　Thumb-2 16 位指令编码格式

16 位指令的操作码部分通过 6 个 bit 进行分类，如表 3-1 所示，每一类指令根据具体情况进行二次编码，因此，Cortex-M3 的指令操作码部分是不等长的，但可以通过多级译码实现，每一级译码的操作码是等长的，由此实现了指令的灵活性和复杂性的均衡。

表 3-1　opcode 分类定义

操作码 opcode	指令或指令类
00xxxx	立即数寻址移位、加法、减法、送数和比较指令
010000	数据处理指令
010001	特殊的数据、分支和交换指令
01001x	寄存器偏移寻址加载指令
0101xx 011xxx 100xxx	单数据加载/存储指令

续表

操作码 opcode	指令或指令类
10100x	PC 相关寻址类指令
10101x	SP 相关寻址类指令
1011xx	16 位的其他指令
11000x	多寄存器存储治理，如 STM，STMIA，STMEA
11001x	多寄存器加载指令，如 LDM，LDMIA，LDMFD
1101xx	有条件转移、SVC 指令
11100x	无条件转移指令

对于 32 位指令，高 16 位的操作码 opcode 的取值为 11101、11110 和 11111，此时处理器会将下一个 16 位和当前 16 位组合成一个 32 位指令，如图 3-2 所示（即 op1 的取值为 01，10 和 11，当 op1 为 00 时，表明是一条 16 位指令）。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	op1		op2											op															

图 3-2　Thumb-2 32 位指令的编码格式

两个寄存器数据的逻辑与 ADD 和逻辑或 ORR 指令编码如图 3-3 所示，从表 3-1 中可以看出逻辑与和逻辑或属于数据处理指令类，数据处理指令又采用了 4 个比特进行了二次编码。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0	0	Rm			Rdn		

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	0	0	Rm			Rdn		

图 3-3　逻辑与和逻辑或的 16 位指令编码

【**思考题**：Thumb2 指令编码是否违反了 RISC 设计思想?】

3.1.2　指令格式

Cortex-M3 不支持 ARM 指令集，支持的指令集包括 ARMv6 的大部分 16 位 Thumb 指令和 ARMv7 的 Thumb-2 指令集。Thumb-2 指令集是一个 16/32 位混合的指令系统。

指令的一般格式如下：

```
<opcode> {<cond> }{S}{.N|.W} <Rd> ,<Rn> {,<operand2> }
```

- opcode 是操作码，如 ADD、LDR 和 STR 等，规定所执行的具体操作；

- cond 是可选的条件码，如 EQ、NE 和 CS 等，规定指令执行所满足的条件，条件码的说明见表 3-2；
- S 是可选后缀，若指定 S，则需要根据指令执行结果去更新程序状态寄存器 xPSR 相应的标志位；
- . N 表示 16 位指令，. W 表示 32 位指令，默认为 16 位指令；
- Rd 是目标寄存器；
- Rn 是存放第 1 个操作数的寄存器；
- operand2 是第 2 个操作数。

这里，符号{ }和＜ ＞的含义如下：

- { }表示可选的，例如{＜cond＞}表示条件码是可选的，可以有条件码也可无条件码；
- ＜ ＞表示必需的，例如＜Rd＞表示必须有目标寄存器。

表 3-2　条件码定义

后　　缀	标　　志	含　　义
EQ	Z=1	相等
NE	Z=0	不相等
CS or HS	C=1	无符号数大于或等于
CC or LO	C=0	无符号数小于
MI	N=1	负的
PL	N=0	正的或为 0
VS	V=1	溢出
VC	V=0	无溢出
HI	C=1 and Z=0	无符号数大于
LS	C=0 or Z =1	无符号数小于或等于
GE	N=V	有符号数大于或等于
LT	N!=V	有符号数小于
GT	Z =0 and N=V	有符号数大于
LE	Z =1 or N!=V	有符号数小于或等于
AL		无条件执行

条件执行是 ARM 处理器的一个优化程序速度的典型方式，可以减少不必要的跳转。如图 3-4 所示的 C 语言代码的 ARM 汇编结果：

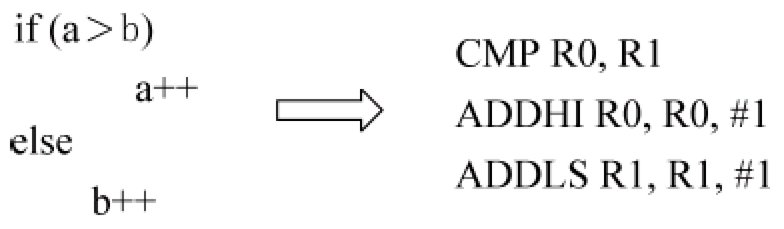


图 3-4　条件执行汇编指令

3.1.3　寻址方式

寻址方式是根据指令中给出的操作数字段来实现寻找真实操作数的方式，Cortex-M3 支持 8 种寻址方式。

1）寄存器寻址

操作数的值在寄存器中，指令中的地址码字段给出的是寄存器编号，寄存器的内容是操作数，指令执行时直接取出寄存器值操作，例如：

```
MOV   R1,R2          ;R1←R2
SUB   R0,R1,R2       ;R0←R1-R2
```

2）立即数寻址

数据就包含在指令当中，立即寻址指令的操作码字段后面的地址码部分就是操作数本身，取出指令也就取出了可以立即使用的操作数（也称为立即数）。立即数要以“#”为前缀，表示十六进制数值时以“0x”表示，例如：

```
ADD   R0,R0,#1       ;R0←R0+1
MOV   R0,#0xff00     ;R0←0xff00
```

3）寄存器移位寻址

寄存器移位寻址是把第 2 个寄存器操作数移位之后送给第 1 个操作数，例如：

```
MOV   R0,R2,LSL #3          ;R2 的值左移 3 位,结果放入 R0,即 R0=R2X8。
AND   R1,R1,R2,LSL R3       ;R2 的值左移 R3 位,然后和 R1 相与操作,结果放入 R1。
```

可采用的移位操作如下：

- LSL：逻辑左移（Logical Shift Left），寄存器中字的低端空出的位补 0。
- LSR：逻辑右移（Logical Shift Right），寄存器中字的高端空出的位补 0。
- ASR：算术右移（Arithmetic Shift Right），移位过程中保持符号位不变，即如果源操作数为正数，则字的高端空出的位补 0，否则补 1。
- ROR：循环右移（Rotate Right），由字的低端移出的位填入字的高端空出的位。
- RRX：带扩展的循环右移（Rotate Right extended by 1 place），操作数右移一位，高端空出的位用进位标志 C 的值填充。

4）寄存器间接寻址

指令中的地址码给出的是一个通用寄存器编号，所需要的操作数保存在该寄存器的值作为地址的存储单元中，即寄存器为操作数的地址指针，操作数存放在存储器中，例如：

```
LDR   R0,[R1]   ;R0←[R1](将 R1 的值作为地址,取出此地址中的数据保存在 R0 中)
STR   R0,[R1]   ;[R1]←R0(将 R0 的值写入到以 R1 的值为地址的存储器空间中)
```

5）变址寻址

变址寻址是将基址寄存器的内容与指令中给出的偏移量相加，形成操作数的有效地址，

变址寻址用于访问基址附近的存储单元，常用于查表，数组操作，外设控制器的内部寄存器访问等，例如：

```
LDR  R2,[R3,#4]     ;R2←[R3+4](将 R3 的数值加 4 作为地址,取出此地址的数值保存在 R2 中)
STR  R1,[R0,#-2]    ;[R0-2]←R1(将 R0 的数值减 2 作为地址,把 R1 中的内容保存到此地址的
                    ;存储单元中)
```

6）多寄存器寻址

采用多寄存器寻址方式，一条指令可以完成多个寄存器值的传送，这种寻址方式用一条指令最多可以完成 16 个寄存器值的传送，例如：

```
LDMIA  R0,{R1,R2,R3,R5}   ;将 R0,R0+4,R0+8,R0+12 地址处的数据分别送到寄存器 R1,R2,
                          ;R3 和 R5 中,R0 的值保持不变
STMIA  R0!,{R1-R7}        ;将 R1~R7 的数据保存到存储器中,存储器指针 R0 在保存第一个值
;之后增加,增长方向为向上增长,即 R1~R7 的值存储在 R0,R0+4,R0+8,R0+12,R0+16,R0+20,R0
;+24 的地址中,指令执行完后,R0 的值变成 R0+24
STMDA  R0!,{R1-R7}        ;将 R1~R7 的数据保存到存储器中,存储器指针 R0 在保存第一个值
;之后减少,增长方向为向下增长,即 R1~R7 的值存储在 R0-24,R0-20,R0-16,R0-12,R0-8,R0-4,
;R0 的地址中,指令执行完后,R0 的值变成 R0-24
```

7）栈寻址

栈寻址是通过栈指针 R13 进行数据读写的方式，如 POP 和 PUSH 操作等，其数据存储和加载的地址由 SP 寄存器隐含给出，例如：

```
PUSH  {R0-R3}             ;将 R0,R1,R2,R3 四个寄存器的值压入栈中
POP    {R0-R2}            ;将栈顶的数据依次读取到 R0,R1,R2 中
LDMIA  SP!,{R1,R2,R3,R5}  ;将栈顶的数据依次读入到 R1,R2,R3,R5 中
```

8）相对寻址

相对寻址是变址寻址的一种变通，由程序计数器 PC 提供基准地址，指令中的地址码字段作为偏移量，两者相加后得到的地址即为操作数的有效地址，例如：

```
    LDR  R2,[PC,#4]     ;R2←[PC+4](将 PC 加 4 作为地址,取出此地址的数保存在 R2 中)
    BL   ROUTE1         ;调用到 ROUTE1 子程序,等价于 LDR  PC,[PC,#6]
    BEQ  LOOP           ;条件跳转到 LOOP 标号处,等价于 LDR  PC,[PC,#2]
LOOP
    MOV R2,#2
ROUTE1
    MOV R1,#3
```

3.1.4　数据传送指令

最基本的数据传送指令是寄存器间的数据传送，此外，还包括立即数加载到寄存器，特殊寄存器的读写指令等。数据传送指令包括 MOV、MVN、MRS 和 MSR 等，具体指令格式

及其功能如表 3-3 所示。

表 3-3　数据传送指令

示　　例	功 能 描 述
MOV <Rd>, #<immed_8>	将 8 位立即数传到目标寄存器
MOV <Rd>, <Rn>	将寄存器值传给低目标寄存器
MVN <Rd>, <Rm>	寄存器值取反后传给目标寄存器
MOV{S}.W <Rd>, #<immed_12>	将 12 位立即数传送到寄存器中
MOV{S}.W <Rd>, <Rm>{,<shift>}	将移位后的寄存器值传到寄存器
MOVT.W <Rd>, #<immed_16>	将 16 位立即数传送到寄存器的高半字[31:16]
MOVW.W <Rd>, #<immed_16>	16 位立即数传到寄存器的低半字[15:0],将高半字[31:16]清零
MRS <Rd>, <SReg>	读特殊功能寄存器 SReg 到寄存器 Rd
MSR <SReg>, <Rn>	写 Rn 到特殊功能寄存器 SReg

例如:

```
MOV   R0, #8                 ;R0=8
MOV   R1,  R0                ;R1=R0=8
MVN   R2,  R1                ;R2=0xFFFFFFF7
MOV.W   R4,  R0 LSL, #2      ;R4=32
MOVW.W   R1,#0x1234          ;R1 =0x1234
MOVT.W   R1,#0x5678          ;R1 =0x56781234
MRS   R1  APSR               ;将 R1 的值写入到状态寄存器 APSR
```

3.1.5　存储器访问指令

存储器访问指令包括存储器寄存器传输指令 LDR、STR;多寄存器加载指令 LDM、多寄存器存储指令 STM 以及压栈指令 PUSH 和出栈指令 POP 等。

1) 加载指令 LDR 和存储指令 STR

LDR 实现从存储器中加载操作数到寄存器,STR 实现从寄存器存储数据到存储器。LDR 和 STR 的语法格式为:

语法格式:

```
op {type}{cond} Rt, [Rn{, #offset}]      ;立即偏移
op {type}{cond} Rt, [Rn, #offset]!       ;前变址
op {type}{cond} Rt, [Rn], #offset        ;后变址
```

这里,op 是 LDR 或 STR。type 指定传送的是字节还是半字,缺省为传送一个字。cond 是可选条件码,Rt 是指定的加载或存储的寄存器,Rn 是存储器地址存放的寄存器,

offset 是偏移量。

type 可以是：

- B：传送无符号的字节；
- SB：传送有符号的字节；
- H：传送无符号的半字；
- SH：传送有符号的半字。

LDR 和 STR 指令进行数据加载和存储时，涉及三种寻址方式，分别为立即偏移的基址变址寻址方式、前变址的基址变址寻址方式和后变址的基址变址寻址方式。这里，基址寄存器为 Rn。

- 立即偏移的基址变址寻址方式，例如 LDR R0，[R1，#4]，这种寻址方式是将基址寄存器 R1 的内容和偏移量 4 相加形成操作数的有效地址；操作完成后，基址寄存器 R1 的内容不变。
- 前变址的基址变址寻址方式，例如 LDR R0，[R1，#4]!，这种寻址方式是将基址寄存器 R1 的内容和偏移量 4 相加形成操作数的有效地址；操作完成后，基址寄存器 R1 的内容更新为新的地址，即 R1＝R1＋4。
- 后变址的基址变址寻址方式，例如 LDR R0，[R1]，#4，这种寻址方式直接将基址寄存器 R1 的内容作为操作数的有效地址；操作完成后，基址寄存器 R1 的内容更新为原有内容和偏移量之和，即 R1＝R1＋4。

使用举例：

```
① LDR R6, [R10]
```

存储器操作数是寄存器间接寻址方式，直接将 R10 寄存器的内容作为操作数的有效地址，该指令实现将有效地址的存储器操作数加载到 R6 寄存器；

```
② LDRNE R6, [R5, #960]!
```

这是带条件码的指令，当程序状态寄存器的 Z 标志位为 0 时才执行该指令。存储器操作数是前变址的基址变址寻址方式，将 R5 寄存器的内容和偏移量 #960 相加作为操作数的有效地址，该指令实现将有效地址的存储器操作数加载到 R6 寄存器；操作完成后，R5 寄存器的内容增加 960。

```
③ STRH R6, [R4], #4
```

这是一个无符号半字的存储指令。存储器操作数是后变址的基址变址寻址方式，直接将 R4 寄存器的内容作为操作数的有效地址，该指令实现将有效地址的存储器操作数（无符号半字）加载到 R6 寄存器；操作完成后，R4 寄存器的内容增加 4。

如果需要对双字进行存取，指令的格式为：

```
opD {cond} Rt, Rt2, [Rn{, #offset}]      ;立即偏移，双字指令
opD {cond} Rt, Rt2, [Rn, #offset]!       ;前变址，双字指令
opD {cond} Rt, Rt2, [Rn], #offset        ;后变址，双字指令
```

其中，op 为 STM 或 LDR，Rt 和 Rt2 为双字存取的寄存器，Rn 为地址寄存器，例如：STRD R1，R2，[R8]，#-16，这是一个双字存储指令。存储器操作数是后变址的基址变址寻址方式，将 R1 的内容存储到 R8 寄存器所指定的有效地址中，将 R2 的内容存储到 R8 寄存器的内容+4 所指定的有效地址中；操作完成后，R8 寄存器的内容减少 16。存储器访问指令的指令格式和功能详见表 3-4。

表 3-4　存储器数据加载和存储指令

示　　例	功 能 描 述
LDRB Rd，[Rn，#offset]	从地址 Rn+offset 处读取一个字节送到 Rd
LDRH Rd，[Rn，#offset]	从地址 Rn+offset 处读取一个半字送到 Rd
LDR Rd，[Rn，#offset]	从地址 Rn+offset 处读取一个字送到 Rd
LDRD Rd1，Rd2，[Rn，#offset]	从地址 Rn+offset 处读取一个双字(64 位整数)送到 Rd1(低 32 位)和 Rd2(高 32 位)中
STRB Rd，[Rn，#offset]	把 Rd 中的低字节存储到地址 Rn+offset 处
STRH Rd，[Rn，#offset]	把 Rd 中的低半字存储到地址 Rn+offset 处
STR Rd，[Rn，#offset]	把 Rd 中的低字存储到地址 Rn+offset 处
STRD Rd1，Rd2，[Rn，#offset]	把 Rd1(低 32 位)和 Rd2(高 32 位)表达的双字存储到 Rn+offset 处
LDR.W Rd，[Rn，#offset]!	字的带预索引加载
LDRB.W Rd，[Rn，#offset]!	字节的带预索引加载(不扩展符号位)
LDRH.W Rd，[Rn，#offset]!	半字的带预索引加载(不扩展符号位)
LDRD.W Rd1，Rd2，[Rn，#offset]!	双字的带预索引加载(不扩展符号位)
LDRSB.W Rd，[Rn，#offset]! LDRSH.W Rd，[Rn，#offset]!	字节/半字的带预索引加载，并且在加载后执行带符号扩展成 32 位整数
STR.W Rd，[Rn，#offset]!	字/字节/半字/双字的带预索引存储
STRB.W Rd，[Rn，#offset]!	字节的带预索引存储
STRH.W Rd，[Rn，#offset]!	半字的带预索引存储
STRD.W Rd1，Rd2，[Rn，#offset]!	双字的带预索引存储

2）批量加载指令 LDM 和批量存储指令 STM

LDM 实现由基址寄存器指示的一片连续存储器中的数据批量加载到多个寄存器，STM 实现将多个寄存器中的数据批量存储到由基址寄存器指示的一片连续存储器。

语法格式：

```
op {addr_mode} {cond} Rn{!}, reglist
```

这里，op 是 LDM 或 STM，addr_mode 是地址变化模式，cond 是可选条件码，Rn 是基址寄存器，! 为可选的回写后缀，若选用该后缀，则数据传送完毕后，将最后的地址写入基址寄

存器,reglist 是多个数据加载或存储的寄存器列表。

addr_mode 可以取下列值:

- IA(Increment After)意味着在每一次访问之后地址递增,如图 3-5(a)所示。
- IB(Increment Before)意味着每一次访问之前地址递增,如图 3-5(b)所示。
- DA(Decrement After)意味着每一次访问之后地址递减,如图 3-5(c)所示。
- DB(Decrement Before)意味着在每一次访问之前地址递减,如图 3-5(d)所示。

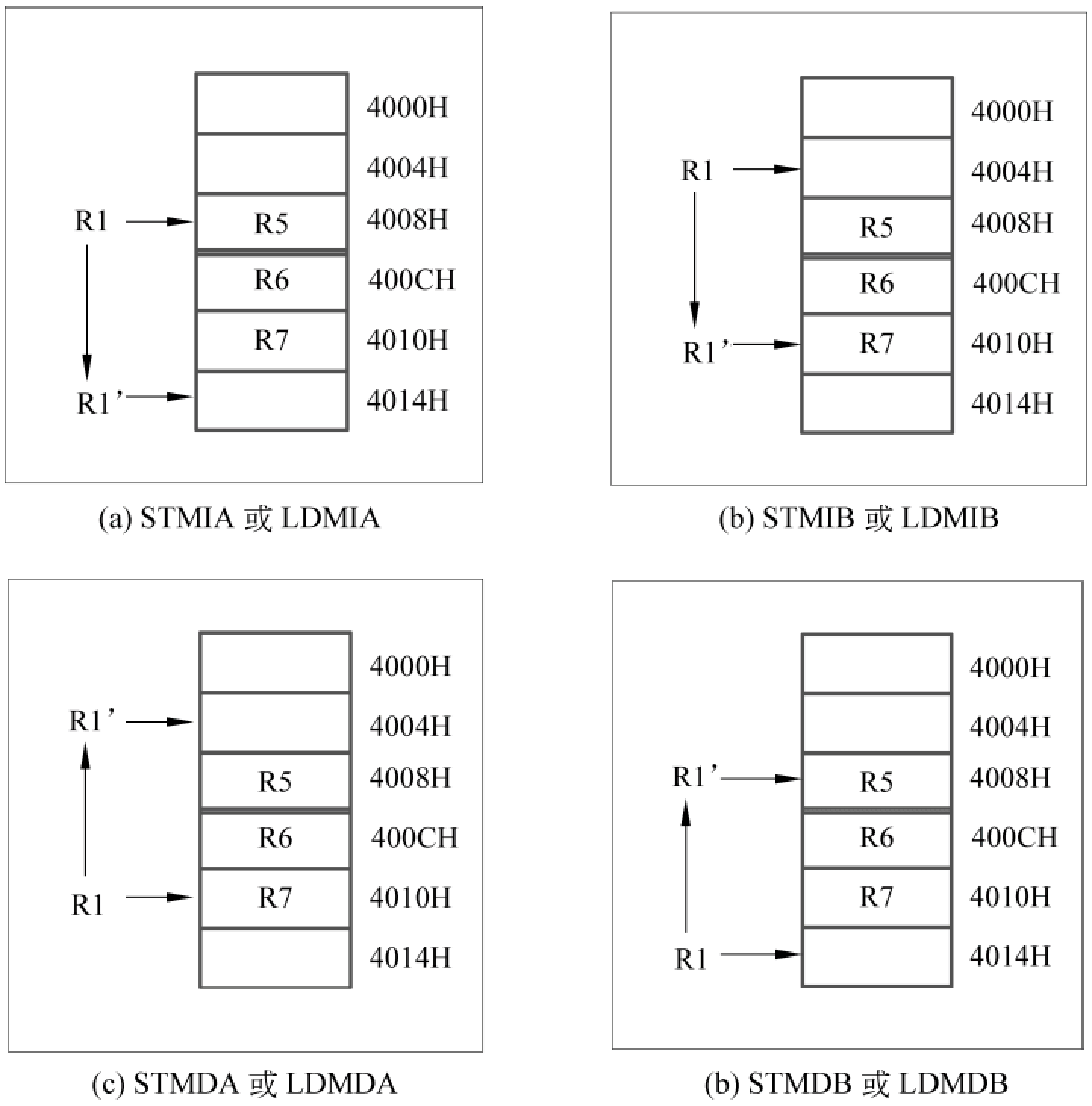


图 3-5　批量加载和批量存储指令中的地址模式

指令举例如下:

① LDM R8, {R0,R2,R9}

该指令实现将 R8 指示的存储器的连续内容加载到三个寄存器 R0、R2 和 R9。R8 的值保持不变。

② STMDB R1!,{R3-R6,R11,R12}

该指令实现将 R3-R6、R11 和 R12 寄存器中的内容存储到 R1 指示的存储器中,并将最后一个存储单元的地址回写到 R1 寄存器。

在多寄存器存储指令中,基址寄存器不能是 PC,寄存器列表中不能含有 SP;若是 STM 指令,寄存器列表一定不能含有 PC;若是 LDM 指令,若寄存器列表中含有 LR 则一定不能

含有 PC；如果增加回写后缀，则寄存器列表中一定不能含有基址寄存器。多寄存器存储指令的指令格式和功能详见表 3-5。

表 3-5　多寄存器访存指令

示　例	功 能 描 述
LDMIA Rd!，{寄存器列表}	16 位指令，从 Rd 处读取多个字，并依次送到寄存器列表中的寄存器。每读一个字后 Rd 自增一次
STMIA Rd!，{寄存器列表}	16 位指令，存储寄存器列表中各寄存器的值依次存储到 Rd 给出的地址。每存一个字后 Rd 自增一次
LDMIA. W Rd!，{寄存器列表}	32 位指令，从 Rd 处读取多个字，并依次送到寄存器列表中的寄存器。每读一个字后 Rd 自增一次
LDMDB. W Rd!，{寄存器列表}	32 位指令，从 Rd 处读取多个字，并依次送到寄存器列表中的寄存器。每读一个字前 Rd 自减一次
STMIA. W Rd!，{寄存器列表}	32 位指令，依次存储寄存器列表中各寄存器的值到 Rd 给出的地址。每存一个字后 Rd 自增一次
STMDB. W Rd!，{寄存器列表}	32 位指令，存储多个字到 Rd 处。每存一个字前 Rd 自减一次

3）压栈指令 PUSH 和出栈指令 POP

当栈指针指向最后压入栈的数据时，称为满栈（Full Stack）；当栈指针指向下一个将要放入数据的空位置时，称为空栈（Empty Stack）。压栈时，栈由低地址向高地址生长时，称为递增栈（Ascending Stack），栈由高地址向低地址生长时，称为递减栈（Descending Stack）。由此可以组合四种栈模型：递增满栈、递增空栈、递减满栈、递减空栈。Cortex-M3 使用的是向下生长的满栈模型。栈指针 SP 指向最后一个被压入栈的 32 位数值。在下一次压栈时，SP 先自减 4，再存入新的数值，如图 3-6 所示。

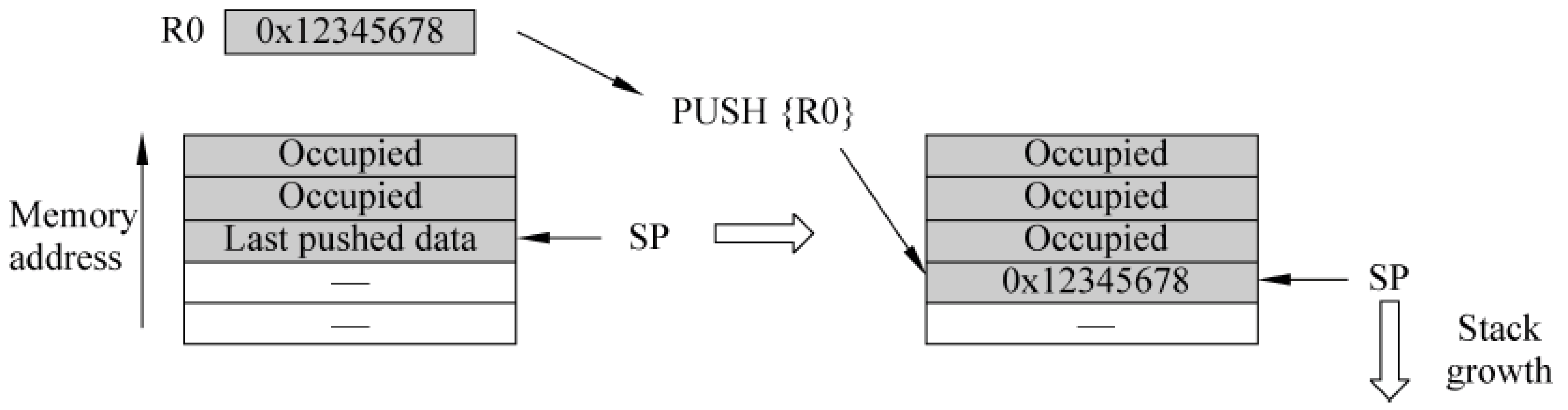


图 3-6　压栈操作过程

出栈操作刚好相反：先从 SP 指针处读出上一次被压入的值，再把 SP 指针自增 4，如图 3-7 所示。

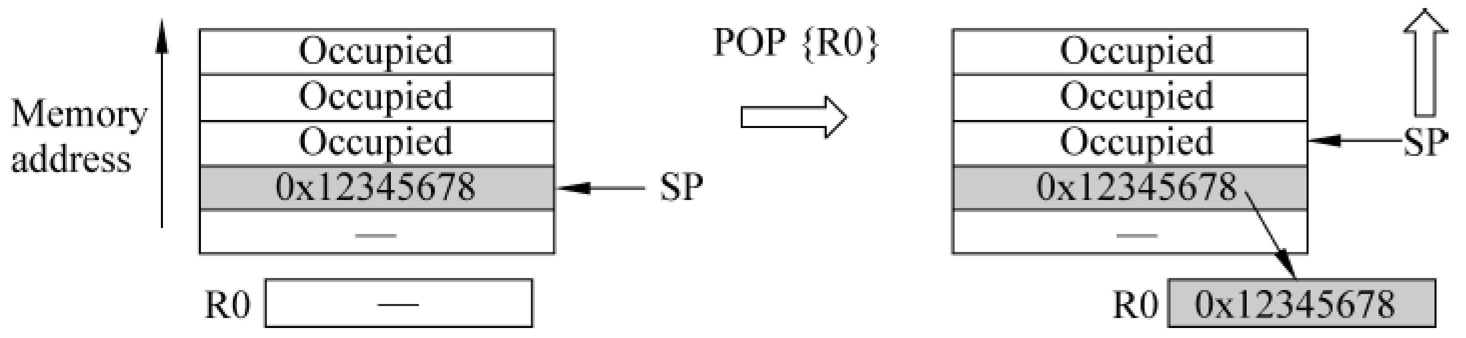


图 3-7　出栈操作过程

PUSH 实现以递减满栈方式的压栈操作；POP 实现以递减满栈方式的出栈操作。因此，PUSH 相当于指令 STMDB 的功能，POP 相当于指令 LDMIA 的功能。

语法格式：

```
PUSH {cond} reglist
POP  {cond} reglist
```

这里，cond 是可选条件码；reglist 是压栈或出栈需要的寄存器列表。

寄存器列表中一定不能含有 SP；对于 PUSH 指令，寄存器列表一定不能含有 PC；对于 POP 指令，如果寄存器列表中含有 LR，则一定不能含有 PC。

使用举例：

```
① PUSH {R0, R4-R7}
```

该指令实现将寄存器列表 R0、R4、R5、R6 和 R7 中数据压入栈。

```
② POP {R0,R6,PC}
```

该指令实现将寄存器列表 R0、R6 和 PC 中的数据弹出栈。

3.1.6 算术运算指令

算术运算指令主要包括加减乘除操作，其指令格式和功能如表 3-6 所示。

表 3-6 算术运算指令

示例		功能描述
ADD Rd, Rn, Rm	; Rd = Rn+Rm	常规加法
ADD Rd, Rm	; Rd += Rm	
ADD Rd, #imm	; Rd += imm	im8(16 位指令)或 im12(32 位指令)
ADC Rd, Rn, Rm	; Rd = Rn+Rm+C	带进位的加法， Im8 或 im12
ADC Rd, Rm	; Rd += Rm+C	
ADC Rd, #imm	; Rd += imm+C	
SUB Rd, Rn	; Rd -= Rn	常规减法
SUB Rd, Rn, #imm3	; Rd = Rn-imm3	
SUB Rd, #imm8	; Rd -= imm8	
SUB Rd, Rn, Rm	; Rd = Rn-Rm	
SBC Rd, Rm	; Rd -= Rm+C	带借位的减法
SBC.W Rd, Rn, #imm12	; Rd = Rn-imm12-C	
SBC.W Rd, Rn, Rm	; Rd = Rn-Rm-C	

续表

示　　例		功 能 描 述
RSB.W Rd，Rn，#imm12	；Rd ＝ imm12－Rn	反向减法
RSB.W Rd，Rn，Rm	；Rd ＝ Rm－Rn	
MUL Rd，Rm	；Rd ＊＝ Rm	常规乘法
MUL.W Rd，Rn，Rm	；Rd ＝ Rn＊Rm	
MLA Rd，Rm，Rn，Ra	；Rd ＝ Ra＋Rm＊Rn	乘加
MLS Rd，Rm，Rn，Ra	；Rd ＝ Ra－Rm＊Rn	乘减
UDIV Rd，Rn，Rm	；Rd ＝ Rn/Rm	无符号除法，硬件支持的除法，余数被丢弃
SDIV Rd，Rn，Rm	；Rd ＝ Rn/Rm	带符号除法，硬件支持的除法，余数被丢弃
SMULL RL，RH，Rm，Rn	；[RH：RL]＝ Rm＊Rn	带符号的 64 位乘法
SMLAL RL，RH，Rm，Rn	；[RH：RL]＋＝ Rm＊Rn	
UMULL RL，RH，Rm，Rn	；[RH：RL]＝ Rm＊Rn	无符号的 64 位乘法
UMLAL RL，RH，Rm，Rn	；[RH：RL]＋＝ Rm＊Rn	

1）ADD、ADC、SUB、SBC 和 RSB

- ADD 实现第 2 个操作数 Operand2 或立即数 imm 与第 1 个操作数 Rn 相加，并将结果送至 Rd 目标寄存器。
- ADC 实现第 2 个操作数 Operand2 与第 1 个操作数 Rn 以及进位标志相加，并将结果送至 Rd 目标寄存器。
- SUB 实现第 1 个操作数 Rn 与第 2 个操作数 Operand2 或立即数 imm 相减，并将结果送至 Rd 目标寄存器。
- SBC 实现第 1 个操作数 Rn 与第 2 个操作数 Operand2 相减，再减去 C 条件标志位的反码，并将结果送至 Rd 目标寄存器。
- RSB 实现第 2 个操作数 Operand2 与第 1 个操作数 Rn 相减，并将结果送至 Rd 目标寄存器。

使用举例：

① ADD R2，R1，R3

该指令实现 R1 寄存器和 R3 寄存器的内容相加，并将结果送至 R2 寄存器，即 R2＝R1＋R3。

② SUBS R7，R5，#256

该指令实现 R5 寄存器的内容与立即数＃256 相减，并将结果送至 R7 寄存器，即 R7＝R5-256。同时，根据运算结果影响标志位。

③ RSB R8, R8, #240

该指令实现立即数 240 与 R8 寄存器的内容相减，并将结果送至 R8 寄存器，即 R8＝240-R8。

④ ADCHI R10, R0, R3

当 C＝1 且 Z＝0 成立时，执行该指令。该指令实现将 R0 寄存器与 R3 寄存器的内容以及进位标志相加，并将结果送至 R10 寄存器，即 R10＝R0＋R3＋C。

2）MUL、MLA 和 MLS

- MUL 指令实现将 Rn 寄存器和 Rm 寄存器的内容相乘，并将结果存入 Rd 寄存器。
- MLA 指令实现将 Rn 寄存器和 Rm 寄存器的内容相乘，并将乘法结果和 Ra 寄存器的内容相加，再将最终结果存入 Rd 寄存器。
- MLS 指令实现将 Rn 寄存器和 Rm 寄存器的内容相乘，并将 Ra 寄存器中的内容与乘法结果相减，再将最终结果存入 Rd 寄存器。

使用举例：

① MUL R7, R2, R5

该指令实现将 R2 寄存器的内容和 R5 寄存器的内容相乘，然后将乘法结果存入 R10 寄存器，即 R10＝R2 * R5。

② MLA R9, R3, R4, R6

该指令实现将 R3 寄存器的内容和 R4 寄存器的内容相乘，然后将乘法结果再与 R6 寄存器内容相加，最后将计算结果存入 R9 寄存器，即 R9＝R3 * R4＋R6。

③ MLS R10, R5, R6, R7

该指令实现将 R5 寄存器的内容和 R6 寄存器的内容相乘，然后将 R7 寄存器的内容与乘法结果相减，最后将计算结果存入 R10 寄存器，即 R10＝R7-R5 * R6。

3）SDIV 和 UDIV

- SDIV 实现有符号整数相除，将 Rn 寄存器的内容与 Rm 寄存器的内容相除，计算结果存入 Rd 寄存器。
- UDIV 实现无符号整数相除，将 Rn 寄存器的内容与 Rm 寄存器的内容相除，计算结果存入 Rd 寄存器。

除法指令同样不能使用 SP 或 PC，对条件标志位没有影响。

使用举例：

① SDIV R0, R1, R2

该指令实现 R1 寄存器内容与 R2 寄存器内容的有符号数相除，计算结果存入 R0 寄存器，即 R0＝R1/R2。

② UDIV R7, R7, R3

该指令实现 R1 寄存器内容与 R2 寄存器内容的有无号数相除，计算结果存入 R7 寄存器，即 R7=R7/R3。

3.1.7　逻辑运算指令

逻辑运算指令包括与、或、非基本操作及其扩展，其指令格式及功能描述见表 3-7 所示。

表 3-7　逻辑运算指令

示　　例		功能描述
AND Rd, Rn AND.W Rd, Rn, #imm12 AND.W Rd, Rm, Rn	; Rd &= Rn ; Rd = Rn & imm12 ; Rd = Rm & Rn	按位与
ORR Rd, Rn ORR.W Rd, Rn, #imm12 ORR.W Rd, Rm, Rn	; Rd |= Rn ; Rd = Rn | imm12 ; Rd = Rm | Rn	按位或
BIC Rd, Rn BIC.W Rd, Rn, #imm12 BIC.W Rd,Rm, Rn	; Rd &= ~Rn ; Rd = Rn & ~imm12 ; Rd = Rm & ~Rn	位清零 Rn 与 Operand2 的反码按位逻辑与
ORN.W Rd, Rn, #imm12 ORN.W Rd, Rm, Rn	; Rd = Rn | ~imm12 ; Rd = Rm | ~Rn	按位或反码
EOR Rd, Rn EOR.W Rd, Rn, #imm12 EOR.W Rd, Rm, Rn	; Rd ^= Rn ; Rd = Rn ^ imm12 ; Rd = Rm ^ Rn	按位异或

- AND 实现第 1 个操作数 Rm 与第 2 操作数 Operand2 的逻辑与操作。
- ORR 实现第 1 个操作数 Rm 与第 2 操作数 Operand2 的逻辑或操作。
- EOR 实现第 1 个操作数 Rm 与第 2 操作数 Operand2 的逻辑异或操作。
- BIC 实现第 1 个操作数 Rm 与第 2 操作数 Operand2 的反码的逻辑与操作，一般可用来清除第 1 个操作数 Rm 的相应位。
- ORN 实现第 1 个操作数 Rm 与第 2 操作数 Operand2 的反码的逻辑或操作。

使用举例：

① AND R1, R1, # 0x00FF

该指令实现 R1 寄存器的内容与立即数#0x00FF 进行逻辑与操作，并将结果送至 R1 寄存器，即实现 R1 寄存器的高 16 位清零。

② BIC R2, R2, # 0x000B

该指令实现 R2 寄存器的内容与立即数#0x000B 的反码进行逻辑与操作，并将结果送至 R2 寄存器，即实现 R2 寄存器的第 0、1 和 3 位的清零。

逻辑运算中逻辑非的操作由送数指令 MVN 实现。

3.1.8 移位和循环指令

移位运算包括逻辑移位、算术移位以及循环移位等特殊形式，具体指令格式和功能见表 3-8。

表 3-8 移位和循环指令

示　例		功能描述
LSL　Rd，Rn，#imm5 LSL　Rd，Rn LSL.W Rd，Rm，Rn	；Rd ＝ Rn<<imm5 ；Rd <<＝ Rn ；Rd ＝ Rm<<Rn	逻辑左移
LSR　Rd，Rn，#imm5 LSR　Rd，Rn LSR.W Rd，Rm，Rn	；Rd ＝ Rn>>imm5 ；Rd >>＝ Rn ；Rd ＝ Rm>>Rn	逻辑右移
ASR　Rd，Rn，#imm5 ASR　Rd，Rn ASR.W Rd，Rm，Rn	；Rd ＝ Rn·>>imm5 ；Rd ·>> ＝ Rn ；Rd ＝ Rm·>> Rn	算术右移
ROR　Rd，Rn ROR.W Rd，Rm，Rn	；Rd >> ＝ Rn ；Rd ＝ Rm >> Rn	循环右移
RRX.W Rd，Rn	；Rd=(Rn>>1)+(C<<31)	带进位的右移一位

- ASR 算术右移指令实现对 Rm 寄存器中的数进行右移 Rn 或 imm5 位操作，左端使用第 31 位值来补充，如图 3-8(a)所示。
- LSL 逻辑左移指令实现对 Rm 寄存器中的数进行左移 Rn 或 imm5 位操作，低位使用 0 填充，如图 3-8(b)所示。
- LSR 逻辑右移指令实现对 Rm 寄存器中的数进行右移 Rn 或 imm5 位操作，左端使用 0 填充，如图 3-8(c)所示。
- ROR 循环右移指令实现对 Rm 寄存器中的数进行右移 Rn 或 imm5 位操作，左端使用右端移出的位来进行填充，如图 3-8(d)所示。
- RRX 带进位的循环右移，左端使用进位标志 C 来进行填充，如图 3-8(e)所示。

使用举例：

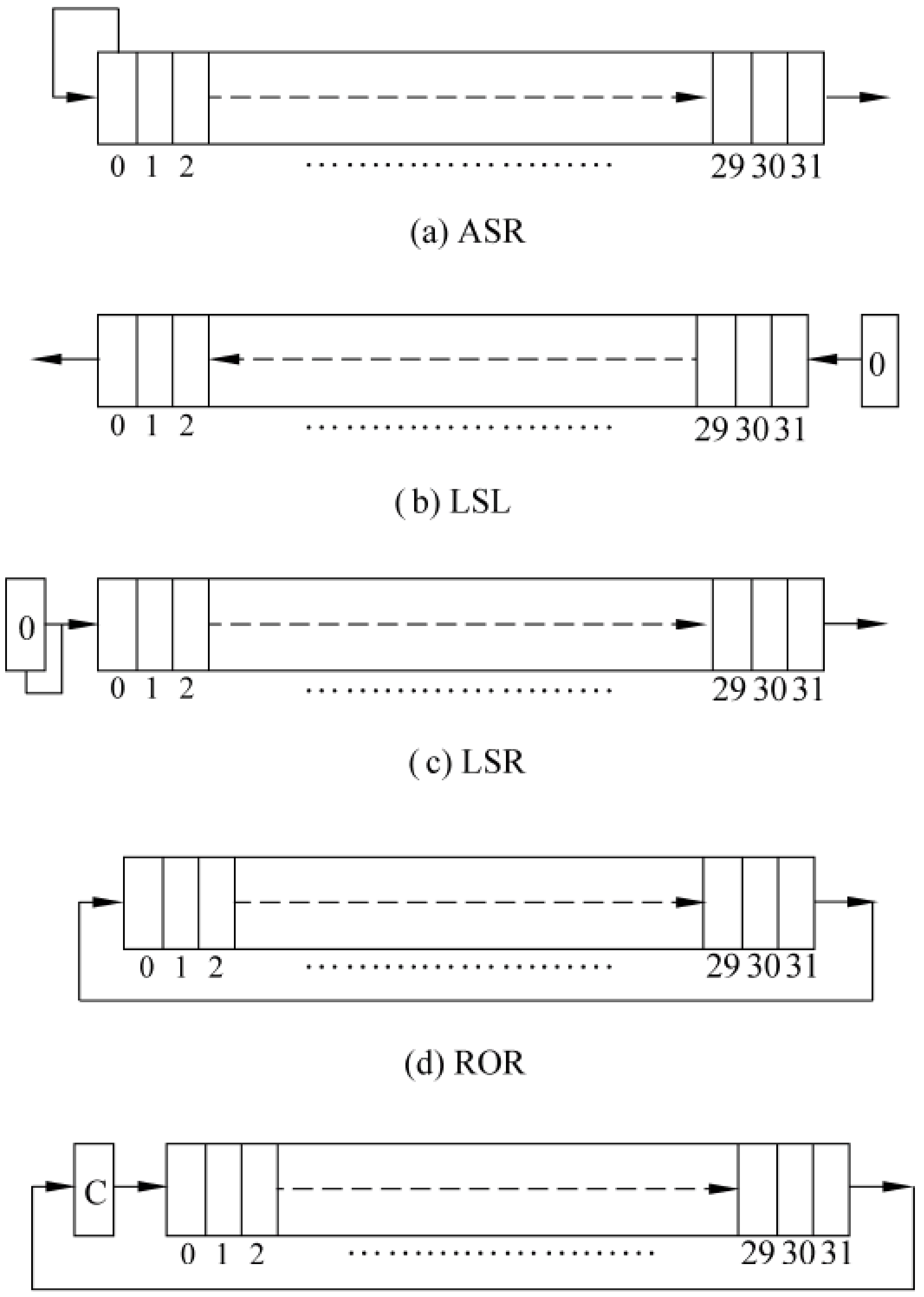
① ASR R1, R2, #9

该指令实现将 R2 寄存中的内容算术右移 9 位(左端使用第 31 位值来填充)，然后将结果填入 R1 寄存器。

② LSLS R3, R4, #3

该指令附加标志位 S，计算结果可能影响标志位。该指令实现将 R4 寄存器逻辑左移 3 位(低位使用 0 值来填充)，然后将结果填入 R3 寄存器。

③ ROR R5, R6, #6



(a) ASR

(b) LSL

(c) LSR

(d) ROR

(e) RRX

图 3-8　移位操作

该指令实现将 R6 寄存器循环右移 6 位(左端使用右端移出的位进行填充)，然后将结果填入 R5 寄存器。

3.1.9　比较指令

CMP 和 CMN 是比较操作指令，其具体指令格式和功能见表 3-9。

表 3-9　比较指令

示　　例	功 能 描 述
CMN ＜Rn＞，＜Rm＞	将 Rm 取二进制补码后再与 Rn 比较
CMP ＜Rn＞，#＜immed_8＞	Rn 与 8 位立即数比较，并根据结果更新标志位的值
CMP ＜Rn＞，＜Rm＞	Rn 与 Rm 比较，并根据结果更新标志位的值
CMN. W ＜Rn＞，#＜ immed_12＞	Rn 与 12 位立即数取补后的值比较
CMN. W ＜Rn＞，＜Rm＞{，＜shift＞}	Rn 与移位后的 Rm 取补后的值比较
CMP. W ＜Rn＞，#＜ immed_12＞	Rn 与 12 位立即数比较
CMP. W ＜Rn＞，＜Rm＞{，＜shift＞}	Rn 与按需移位后的 Rm 比较，Rm 的值不变

CMP 指令实现将 Rn 寄存器的内容减去第 2 个操作数 Operand2，计算结果影响标志位。该指令类似于 SUBS 指令，所不同的是 CMP 指令丢弃减法计算结果，而 SUBS 指令需要保存减法计算结果。

CMN 指令实现将第 2 个操作数加到 Rn 寄存器中，计算结果影响标志位。该指令类似于 ADDS 指令，所不同的是 CMN 指令丢弃加法计算结果，而 ADDS 指令需要保存加法计算结果。

比较指令根据计算结果影响标志位 N、Z、C 和 V。

使用举例：

① CMP R1, #6400

该指令将 R1 寄存器的内容减去立即数#6400，计算结果影响标志位。

② CMN R2, R1

指令将 R1 寄存器的内容加到 R2 寄存器，计算结果影响标志位。

3.1.10 分支控制指令

分支控制指令(Branch and Control Instructions)包括直接跳转指令 B 和 BL、间接跳转指令 BX 和 BLX，具体指令格式和功能见表 3-10.

表 3-10 跳转指令

示　例	功能描述
B<cond> <target address>	按<contd>条件决定是否分支
B<tartet address>	无条件分支
BL <Rm>	带链接分支
B{cond}. W <label>	条件分支
BL. W <label>	带链接的分支
BL. W<c> <label>	带链接的分支(立即数)
B. W <label>	无条件分支

- B 是直接跳转指令，label 可以是 24 位的有符号数，跳转范围是－16～16MB。
- BL 除了可以直接跳转到 label 处外，在跳转前会将 PC 内容保存到 LR 寄存器。
- BX 是间接跳转指令，跳转到的目标地址存放在 Rm 寄存器。
- BLX 也是一个间接跳转指令，同样在跳转前会将 PC 内容保存到 LR 寄存器。

使用举例：

```
① B loop          ;直接跳转到 loop 处;
② BLE ng          ;当 Z=1 或 N!=V,即有符号数小于或等于时,直接跳转到 ng 处;
③ BEQ target1     ;当 Z =1,即相等时,直接跳转到 target1 处
④ BX LR           ;返回函数调用处
```

3.1.11　其他指令

主要包括位操作指令、符号扩展指令、字节交换指令等，指令的具体格式和功能见表 3-11。

表 3-11　其他指令

示　　例	功 能 描 述
BFC. W Rd，Rn，＃＜width＞	位区清零
BFI. W Rd，Rn，＃＜lsb＞，＃＜width＞	将一个寄存器的位区插入另一个寄存器中
SBFX. W Rd，Rn，＃＜lsb＞，＃＜width＞	复制位段，并带符号扩展到 32 位
SBFX. W Rd，Rn，＃＜lsb＞，＃＜width＞	复制位段，并无符号扩展到 32 位
REV. W Rd，Rn	在字中反转字节序
REV16. W Rd，Rn	在高低半字中反转字节序
REVSH. W	在低半字中反转字节序，并做带符号扩展
SXTB Rd，Rm{，＜rotation＞}	Rd ＝ Rm 把带符号字节整数扩展到 32 位
SXTH Rd，Rm{，＜rotation＞}	Rd ＝ Rm 把带符号半字整数扩展到 32 位

1）BFC 和 BFI

- BFC 指令实现 Rd 寄存器中从 1sb 开始的 width 位数的位清零。
- BFI 指令实现 Rn 寄存器中从 0 开始的 width 位拷贝到 Rd 寄存器中从 1sb 开始的 width 位。

使用举例：

① BFC R1，＃8，＃13

该指令实现对 R1 寄存器中从第 8 位开始的 13 位（即到第 20 位）的数据进行清零。

② BFI R2，R3，＃7，＃11

该指令实现将 R3 寄存器中从第 0 位到第 10 位的数据拷贝到 R2 寄存器的第 7 位到第 17 位。

2）SBFX 和 UBFX

- SBFX 指令实现将 Rn 寄存器中从第 1sb 位开始的 width 位抽取出来，然后进行有符号位扩展到 32 位并将结果存入 Rd 寄存器。
- UBFX 指令实现将 Rn 寄存器中从第 1sb 位开始的 width 位抽取出来，然后进行无符号位扩展到 32 位并将结果存入 Rd 寄存器。

使用举例：

① SBFX R1，R2，＃10，＃4

该指令实现将 R2 寄存器中的第 10 位到第 13 位抽取出来并进行有符号扩展到 32 位，

然后将结果存入 R1 寄存器。

② UBFX R3, R4, #9, #10

该指令实现将 R4 寄存器中的第 9 位到第 18 位抽取出来并进行无符号扩展到 32 位，然后将结果存入 R3 寄存器。

3) SXTB、SXTH、UXTB 和 UXTH

- SXTB 指令实现将 Rm 寄存器的低 8 位，或 Rm 寄存器经过循环右移 rotation 位后的低 8 位，有符号扩展到 32 位，然后存入 Rd 寄存器。
- UXTH 指令实现将 Rm 寄存器的低 16 位，或 Rm 寄存器经过循环右移 rotation 位后的低 16 位，无符号扩展到 32 位，然后存入 Rd 寄存器。

使用举例：

① SXTH R1, R2, ROR #16

该指令首先将 R2 寄存器的内容进行循环右移 16 位，然后取出低 16 位的半字并进行有符号扩展到 32 位，最后将结果存入 R1 寄存器。

② UXTB R3, R10

该指令首先取出 R10 寄存器的低 8 位，然后进行无符号扩展到 32 位，最后将结果存入 R3 寄存器。

4) REV、REV16、REVSH

- REV 实现一个字 Rn 的四个字节大小端转换，复制到 Rd 中。
- REV16 实现 Rn 的两个半字内部的大小端转换，复制到 Rd 中。
- REVSH 将 Rn 低半字内的字节反转，再把反转后的值带符号位扩展到 32 位后，复制到 Rd 中。

字节交换指令的原理见图 3-9。

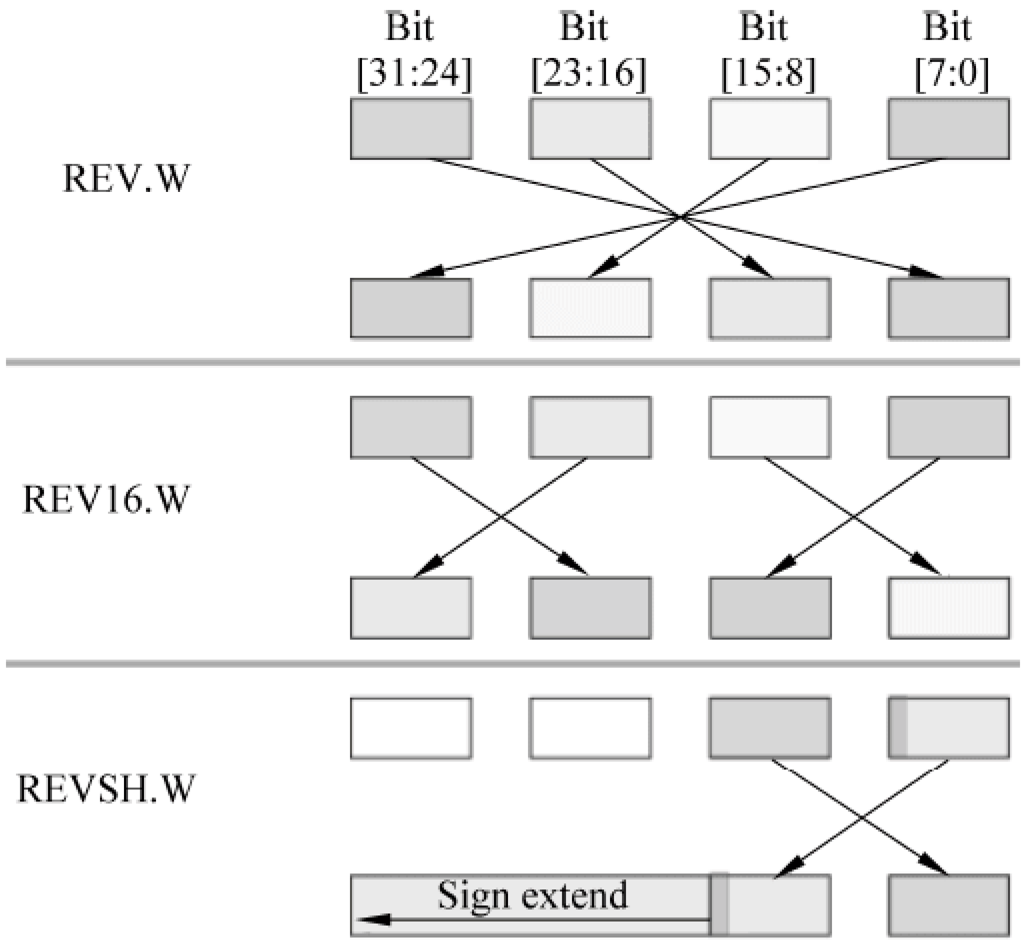


图 3-9 字节交换指令交换顺序

3.2　ARM 汇编器中的伪指令

ARM 汇编语言程序里，有一些特殊指令助记符，这些助记符与指令系统的助记符不同，没有相对应的操作码，通常称这些特殊指令助记符为伪指令。伪指令在源程序中的作用是为完成汇编程序作各种准备工作。有以下几种伪指令：符号定义伪指令、数据定义伪指令、汇编控制伪指令和宏指令。

3.2.1　Thumb 伪指令

1）ADR

小范围的地址读取伪指令。ADR 指令将基于 PC 相对偏移的地址值读取到寄存器中。ADR 伪指令格式如下：

```
ADR register,expr
```

其中，register 为加载的目标寄存器，expr 为地址表达式。偏移量必须是正数并小于 1KB。

ADR 伪指令示例：

```
    ADR R0,TxtTab
    …
TxtTab:
    DCB "ARM7TDMI",0
```

2）LDR

大范围的地址读取伪指令。LDR 伪指令用于加载 32 位的立即数或一个地址值到指定寄存器。在汇编编译源程序时，LDR 伪指令被编译器替换成一条合适的指令。若加载的常数未超出 MOV 范围，则使用 MOV 或 MVN 指令代替 LDR 伪指令，否则汇编器将常量放入文字池，并使用一条程序相对偏移的 LDR 指令从文字池读出常量。LDR 伪指令格式如下：

```
LDR register,=expr/label_expr
```

其中，register 为加载的目标寄存器，expr 是 32 位立即数，label_expr 是基于 PC 的地址表达式或外部表达式。

LDR 伪指令举例如下：

```
LDR R0,=0x12345678        ;加载 32 位立即数 0x12345678
LDR R0,=DATA_BUF+60       ;加载 DATA_BUF 地址+60
⋮
```

```
LTORG                    ;声明文字池
  ⋮
```

文字池一般由 ARM 编译器自动分配，从 PC 到文字池的偏移量必须是正数小于是 1KB。与 Thumb 指令的 LDR 相比，伪指令的 LDR 的参数有“＝”号。

3）NOP

空操作伪指令。NOP 伪指令在汇编时将会将会被代替成 ARM 中的空操作，比如可能为“MOV R8，R8”指令等，可用于延时操作。NOP 伪指令格式如下：

```
NOP
```

3.2.2　符号定义伪指令

符号定义伪指令用于定义汇编程序中的变量、进行变量赋值以及定义寄存器的别名等操作。常见的符号定义伪指令包括：

1）定义全局变量的伪指令 GBLA、GBLL 和 GBLS

语法格式：

```
GBLA(GBLL或 GBLS) 全局变量名
```

GBLA、GBLL 和 GBLS 伪指令用于定义一个 ARM 程序中的全局变量，并将其初始化。其中：

- GBLA 伪指令用于定义一个全局的数字变量，并初始化为 0；
- GBLL 伪指令用于定义一个全局的逻辑变量，并初始化为 F(假)；
- GBLS 伪指令用于定义一个全局的字符串变量，并初始化为空。

使用示例：

```
GBLA T1                  ;定义一个全局的数字变量,变量名为 T1,初始值为 0
GBLL T2                  ;定义一个全局的逻辑变量,变量名为 T2,初始值为 F
GBLS T3                  ;定义一个全局的字符串变量,变量名为 T3,初始值为空
```

2）定义局部变量的伪指令 LCLA、LCLL 和 LCLS

语法格式：

```
LCLA(LCLL或 LCLS) 局部变量名
```

LCLA、LCLL 和 LCLS 伪指令用于定义一个 ARM 程序中的局部变量，并将其初始化。其中：

- LCLA 伪指令用于定义一个局部的数字变量，并初始化为 0；
- LCLL 伪指令用于定义一个局部的逻辑变量，并初始化为 F(假)；
- LCLS 伪指令用于定义一个局部的字符串变量，并初始化为空。

定义局部变量的伪指令的使用方法与定义全局变量的伪指令的使用方法类似。

3) 进行变量赋值的伪指令 SETA、SETL 和 SETS

语法格式:

```
变量名 SETA(SETL 或 SETS) 表达式
```

伪指令 SETA、SETL、SETS 用于给一个已经定义的全局变量或局部变量赋值。其中:

- SETA 伪指令用于给一个数学变量赋值;
- SETL 伪指令用于给一个逻辑变量赋值;
- SETS 伪指令用于给一个字符串变量赋值。

使用示例:

```
LCLL T4             ;定义一个局部的逻辑变量,变量名为 T4
T4 SETL {TRUE}      ;将该逻辑变量赋值为 TRUE
```

3.2.3　数据定义伪指令

数据定义伪指令一般用于为特定的数据分配存储单元并进行初始化。常见的数据定义伪指令包括:

1) 连续分配一片连续的字节存储单元的伪指令 DCB

语法格式:

```
标号 DCB 表达式
```

DCB 伪指令用于分配一片连续的字节存储单元并用伪指令中指定的表达式初始化。其中,表达式可以为 0～255 的数字或字符串。

使用示例:

```
MyName DCB "This is my name."
```

分配一片连续的字节存储单元并初始化,起始地址为 MyName。

2) 连续分配一片连续的半字存储单元的伪指令 DCW(或 DCWU)

语法格式:

```
标号 DCW(或 DCWU) 表达式
```

DCW(或 DCWU)伪指令用于分配一片连续的半字存储单元并用伪指令中指定的表达式初始化。使用 DCW 分配的字存储单元是半字对齐的,而用 DCWU 分配的字存储单元并不严格半字对齐。

使用示例:

```
WTest DCW 1,2,3;
```

分配 3 个连续的半字存储单元并初始化为 1,2,3,起始地址为 Wtest。

3）连续分配一片连续的字存储单元的伪指令 DCD(或 DCDU)

语法格式：

```
标号 DCD(或 DCDU) 表达式
```

DCD(或 DCDU)伪指令用于分配一片连续的字存储单元并用伪指令中指定的表达式初始化。用 DCD 分配的字存储单元是字对齐的，而用 DCDU 分配的字存储单元并不严格字对齐。

使用示例：

```
DTest DCD 4,5,6 ;
```

分配 3 个连续的字存储单元并初始化为 4，5，6，起始地址为 Dtest。

4）分配一片连续的存储单元的伪指令 SPACE

语法格式：

```
标号 SPACE 表达式
```

SPACE 伪指令用于分配一片连续的存储区域并初始化为 0。其中，表达式为要分配的字节数。

使用示例：

```
DataSpace SPACE 10;
```

分配连续 10 个字节的存储单元并初始化为 0，起始地址为 DataSpace。

3.2.4 汇编控制伪指令

汇编控制伪指令用于控制汇编程序的执行流程，常用的汇编控制伪指令包括如下两条：

1）IF、ELSE、ENDIF

语法格式：

```
IF 逻辑表达式
指令序列 1
ELSE
指令序列 2
ENDIF
```

IF、ELSE、ENDIF 伪指令能根据条件的成立与否决定是否执行某个指令序列。当 IF 后面的逻辑表达式为真，则执行指令序列 1，否则执行指令序列 2。其中，ELSE 及指令序列 2 可以没有，此时，当 IF 后面的逻辑表达式为真，则执行指令序列 1，否则继续执行后面的指令。

使用示例：

```
GBLL Flag
```

```
…
IF Flag = TRUE
指令序列 1
ELSE
指令序列 2
ENDIF
```

2）WHILE、WEND

语法格式：

```
WHILE 逻辑表达式
指令序列
WEND
```

WHILE、WEND 伪指令能根据条件的成立与否决定是否循环执行某个指令序列。当WHILE 后面的逻辑表达式为真，则执行指令序列，该指令序列执行完毕后，再判断逻辑表达式的值，若为真则继续执行，一直到逻辑表达式的值为假。

使用示例：

```
GBLA Counter
Counter SETA 3
⋮
WHILE Counter < 10
指令序列
WEND
```

3.2.5 其他常用的伪指令

其他的伪指令包括：

1）定义代码段或数据段伪指令 AREA

AREA 伪指令用于定义一个代码段或数据段，其语法格式如下。

```
AREA 段名 属性 1,属性 2,…
```

使用示例：

```
AREA Init,CODE,READONLY
```

该伪指令定义了一个代码段，段名为 Init，属性为只读。

2）指令标识伪指令 CODE16、CODE32

CODE16 伪指令通知编译器，其后的指令序列为 16 位的 Thumb 指令。

CODE32 伪指令通知编译器，其后的指令序列为 32 位的 ARM 指令。

使用示例：

```
AREA Init,CODE,READONLY
…
CODE32              ;通知编译器其后的指令为 32 位的 ARM 指令
CODE16              ;通知编译器其后的指令为 16 位的 Thumb 指令
…
```

3）指定应用程序入口的伪指令 ENTRY

ENTRY 伪指令用于指定汇编程序的入口点。

使用示例：

```
AREA Init,CODE,READONLY
ENTRY               ;指定应用程序的入口点
…
```

4）指定应用程序结束的伪指令 END

END 伪指令用于通知编译器已经到了源程序的结尾。

使用示例：

```
AREA Init,CODE,READONLY
…
END                 ;指定应用程序的结尾
```

3.3 汇编语言的程序结构

汇编语言程序中,以程序段为单位组织代码。段是相对独立的指令或数据序列,具有特定的名称。段可以分为代码段和数据段,代码段的内容为执行代码,数据段存放代码运行时需要用到的数据。一个汇编程序至少应该有一个代码段,多个段在程序编译链接时最终形成一个可执行文件。

以下是一个汇编语言源程序的基本结构：

```
AREA Init,CODE,READONLY
ENTRY
  LDR R0,=0x3FF5000
  LDR R1,0xFF
  STR R1,[R0]
  LDR R0,=0x3FF5008
  LDR R1,0x01
  STR R1,[R0]
END
```

本例定义了一个名为 Init 的代码段,属性为只读。ENTRY 伪指令标识程序的入口点,接下来为指令序列,程序的末尾为 END 伪指令,该伪指令告诉编译器源文件的结束,每一

个汇编程序段都必须有一条 END 伪指令，指示代码段的结束。多加几个例子说明汇编程序设计。

1）循环累加案例

用汇编实现 1＋2＋3＋…＋10 的累加计算，代码如下。

```
STACK_TOP EQU 0x2000 0200                ;栈地址
AREA Init,CODE,READONLY
ENTRY
DCD STACK_TOP                            ;复位后建立栈指针
DCD START                                ;复位后执行的代码地址
START:
    MOVS R0, #10                         ;初始化
    MOVS R1, #0
;计算 10+9…+1
LOOP:
    ADDS R1, R0                          ;R1=R1+R0
    SUBS R0, #1                          ;R0=R0-1
    BNE LOOP                             ;不为 0 跳转
    LDR R0, =RESULT                      ;获取数据存储区地址
    STR R1, [R0]                         ;结果现在存储到 R1 中了
DEADLOOP:
    B DEADLOOP
;数据区定义
AREA BUF,DATA,READWRITE
RESULT:
    DCD 0                                ;数据存储区
    END
```

2）启动代码分析

启动代码是处理器上电后执行的第一部分代码，启动代码完成程序的栈空间、堆空间以及复位中断和其他中断的中断向量入口设置，最后跳转到 Main 函数执行。根据不同的处理器，CMSIS 库提供不同的启动代码。以下对 STM32L152 系列处理器的启动代码进行分析。

启动代码中首先定义了栈、堆的大小。

```
Stack_Size      EQU     0x00000400                    ;栈大小
AREA    STACK, NOINIT, READWRITE, ALIGN=3            ;定义数据段,8字节对齐
Stack_Mem       SPACE   Stack_Size                    ;开辟栈空间
__initial_sp                                          ;栈顶标号

Heap_Size       EQU     0x00000200                    ;堆大小
AREA    HEAP, NOINIT, READWRITE, ALIGN=3             ;定义数据段
```

```
__heap_base                                 ;堆的起始地址
Heap_Mem       SPACE    Heap_Size           ;开辟堆空间
__heap_limit                                ;堆的结束地址

PRESERVE8                                   ;指示编译器 8字节对齐
THUMB                                       ;指示编译器为 THUMB指令
```

其次,启动程序中定义了所有的中断向量名称及其入口地址,中断向量表在复位时映射到 0 地址。

```
AREA     RESET, DATA, READONLY              ;定义代码段,位于 0地址
;申明三个标号
EXPORT   __Vectors
EXPORT   __Vectors_End
EXPORT   __Vectors_Size
;用 DCD定义一个字存储空间,存放后面符号的地址,这些符号名称在 stm32L1xx_it.c中是一个函
;数名,在该函数中添加代码即可实现中断处理
__Vectors       DCD     __initial_sp        ; 栈顶地址
                DCD     Reset_Handler       ; 复位中断函数
                DCD     NMI_Handler         ; NMI 中断函数
                DCD     HardFault_Handler   ; Hard Fault Handler
                DCD     MemManage_Handler   ; MPU Fault Handler
                DCD     BusFault_Handler    ; Bus Fault Handler
                DCD     UsageFault_Handler  ; Usage Fault Handler
                …
                DCD     TIM6_IRQHandler     ; TIM6
                DCD     TIM7_IRQHandler     ; TIM7
__Vectors_End                ;标记中断向量表的结束地址
__Vectors_Size  EQU __Vectors_End-__Vectors ;计算中断向量表大小
```

启动代码的主程序部分是复位中断处理函数。

```
AREA     |.text|, CODE, READONLY            ;定义代码段
Reset_Handler     PROC                      ;定义复位函数
EXPORT   Reset_Handler       [WEAK]
IMPORT   __main                             ;导入__main函数的地址
IMPORT   SystemInit                         ;导入 SystemInit函数的地址
          LDR     R0, =SystemInit
          BLX     R0                        ;跳转到 SystemInit运行
          LDR     R0, =__main
          BX      R0                        ;跳转到__main函数执行
ENDP
```

其中 EXPORT Reset_Handler [WEAK]是导出 Reset_Handler 标识,[WEAK]用来

表明如果由外部定义的其他 Reset_Handler 函数，则执行其他 Reset_Handler 函数。

SystemInit 函数在 system_stm32l1xx.c 中，其功能是初始化 flash 接口，设置启动时钟。

__main 是系统提供的主程序调用库，__main 函数主要执行的功能包括：

- 加载 RO 和 RW 代码及数据；
- 初始化静态存储区数据为 0；
- 初始化堆栈；
- 跳转到 C 语言的 main 函数执行；
- 处理 main 函数的返回值。

【思考题：__main 和 C 语言的 main 函数是一个函数吗？**】**

Chapter 4

第 4 章　开发板硬件系统及开发环境

【导读】 本章为嵌入式系统开发的基础知识，首先介绍嵌入式硬件最小系统的概念的组成，典型的外围电路原理，然后对嵌入式开发流程、CMSIS 库的结构和功能进行详细阐述。针对嵌入式程序设计中涉及的 C 语言常用知识，如宏定义、Volatile、位与、位或、按位取反、左移、右移、寄存器操作等基础知识，本章进行了简要梳理。本章的目的为建立嵌入式开发的流程，掌握嵌入式 C 开发的基础知识。

4.1　最小系统设计

一个嵌入式处理器自己不能独立工作，必须要加上电源、提供复位和时钟信号，如果嵌入式处理器片内没有存储器，还需要加上片外 Flash、RAM 构成一个系统才能正常工作。嵌入式处理器运行所必需的电路和嵌入式处理器构成了嵌入式处理器的最小系统。系统的调试接口在运行时不是必需的，但在开发时必须要使用，因此调试接口也是最小系统组成之一。

最小系统的基本组成如图 4-1 所示。

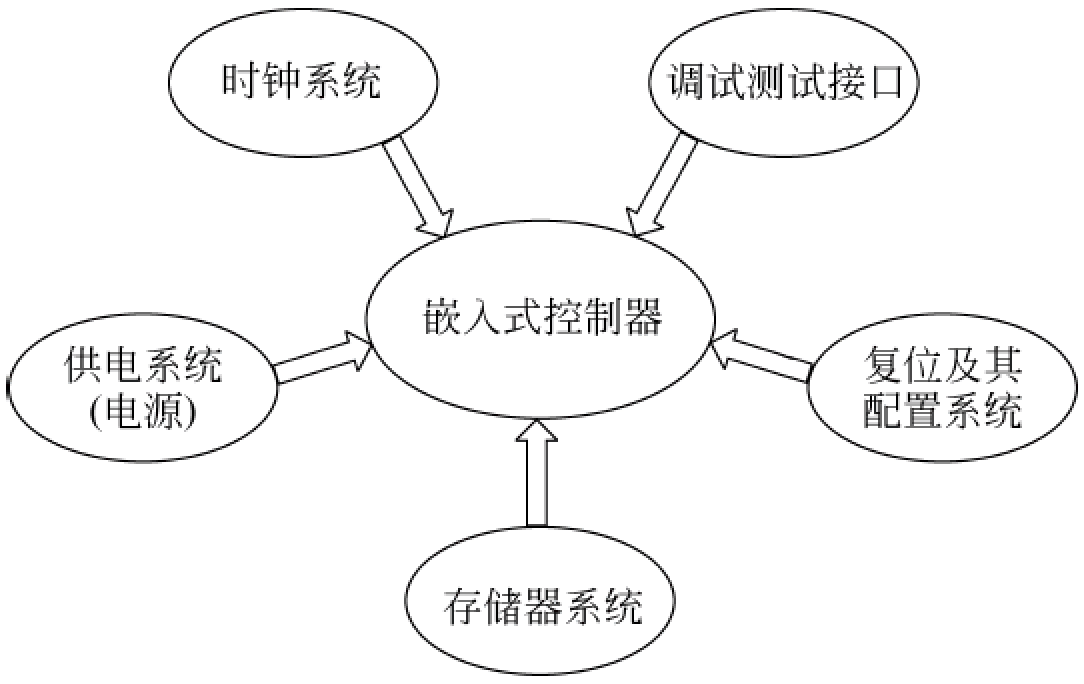


图 4-1　嵌入式系统的最小组成

(1) **供电系统**：电源系统为整个系统提供能量，是系统工作的基础。嵌入式处理器的电源一般采用 5V、3.3V、1.8V 等直流供电，分为数字电源和模拟电源，模拟电源一般用于 AD 采集模块，两个电源在要求不高的情况下可以不分开。为保证处理器供电的稳定性，一

般采用电源芯片对输入电压进行稳压,在设计时需要考虑到电源的功率是否满足系统需求。

(2) **存储系统**:对于大多数嵌入式处理器,其内部都带有片内 Flash 和 RAM,这样外部无需增加存储器,如果内部不带存储器或者容量不能满足系统需求,则需要外扩存储器,一般通过外部总线进行连接。

(3) **复位电路**:嵌入式处理器在上电工作时,需要将处理器的状态和内部寄存器初始化为一个确认的状态,以防止程序运行不能正确执行,因此需要给处理器一个复位信号。复位信号一般要求持续一定时间,在上电时可以通过阻容复位电路给出,也可以通过专用的复位芯片给出。在系统工作过程中,如果碰到电源电压过低、干扰等可能导致嵌入式处理器无法正常工作的情况,复位芯片给出复位信号。

(4) **调试接口**:嵌入式处理器一般带有 JTAG 调试接口,通过 JTAG 调试接口可以控制芯片的运行并获取内部信息。ARM Cortex 系列处理器采用了新的 CoreSight,内核本身不再含有 JTAG 接口。取而代之的是调试访问接口(DAP),由一个在芯片内部实现的调试端口设备(DP)完成,也支持 JTAG 调试方法。

(5) **时钟系统**:嵌入式处理器一般为时序电路,需要时钟信号才能工作,大多数嵌入式处理器内部集成振荡器,作为时钟源;内部时钟一般精确度较低,因此在一些时序要求严格的场合需要外部晶振,此外,对于低功耗应用,嵌入式处理器在很低的功耗下需要外部时钟唤醒,此时需要在电路上增加外部晶振。

4.2 开发板电路原理图

本教材使用的开发板为 ST 的 STM32L-Discovery 开发板,开发板集成一个 STLINK 调试器、两个按键、一个显示屏、一个触摸按键(可作为 4 个按键),并将所有的芯片输入输出引脚引出便于扩展使用。开发板的结构如图 4-2 所示。以下将分别对开发板电路各个部分的原理进行介绍。

4.2.1 电源

STM32L-Discovery 开发板提供 5V 和 3.3V 两个电源,开发板电源可采用 PC USB 口供电,或者通过单独的 5V 或 3.3V 直流电源供电。

如图 4-3 所示,5V 的直流电源经过一个稳压芯片 LD3985M33 后转换为 3.3V,开发板在扩展引脚 EXT_5V 和 EXT_3V 上将 5V 和 3.3V 的两个电源单独引出,可以作为输出电源供给其他电路(电流小于 100mA)。图 4-3 中的二极管 D1 和 D2 对 3.3V 电源和 5V 电源进行了保护,使得开发板也可以在扩展引脚 EXT_5V 和 EXT_3V 上外接 5V 和 3.3V 电源作为输入电源。

为便于电池供电的低功耗应用设计,开发板提供了一个纽扣电池供电电路,如图 4-4 所示可通过板子的跳线进行电源切换。

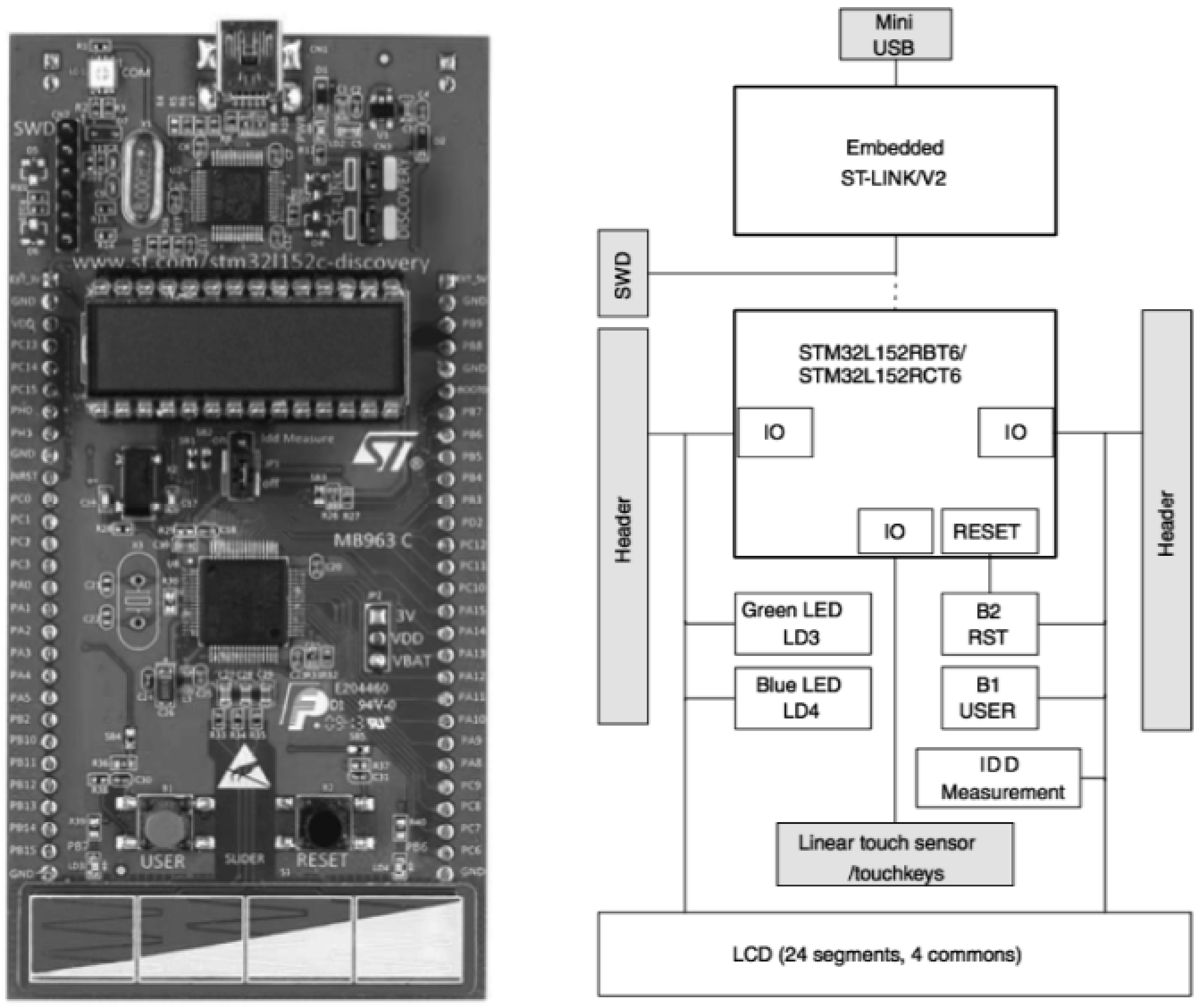


图 4-2　STM32L152-Discovery 开发板

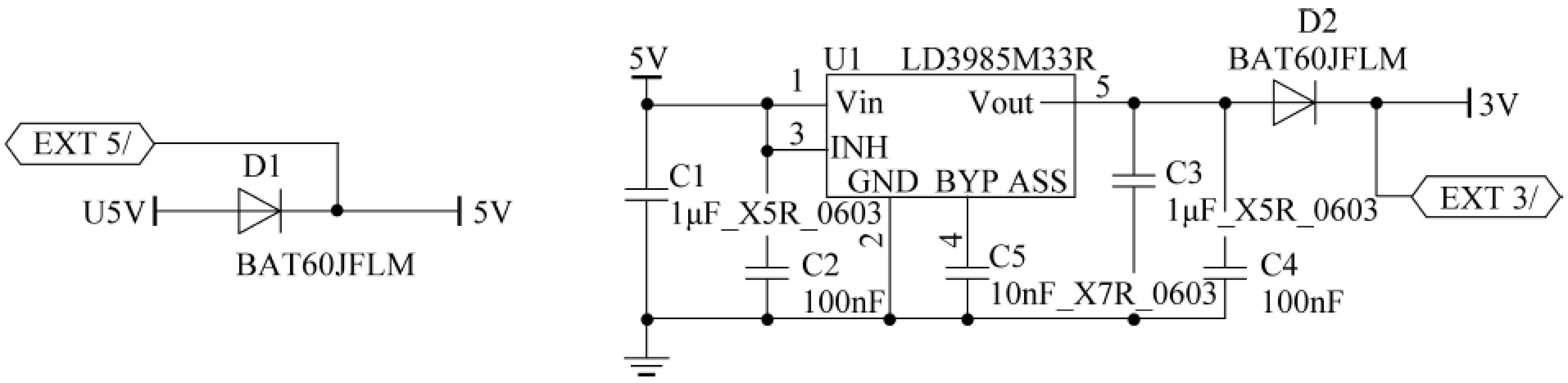


图 4-3　电源电路原理

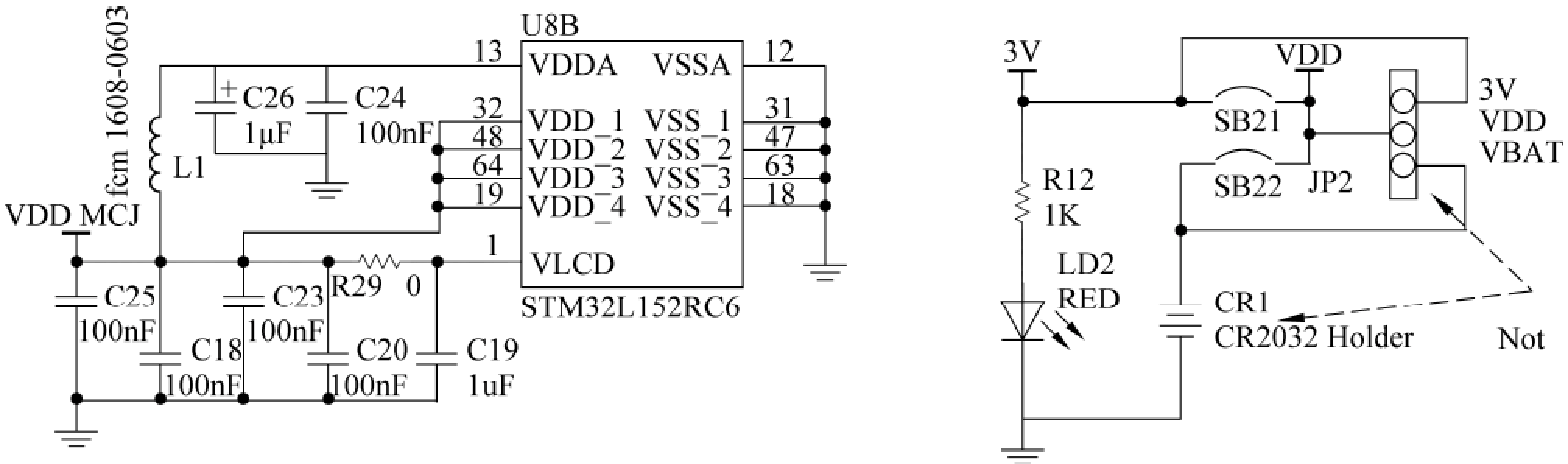


图 4-4　电池切换电路

STM32L152 处理器的电源包括模拟电源 VDDA，数字电源 VDD 和液晶屏电源 VLCD，供电电压为 3.3V。模拟电源用于内部 AD 转换器，LCD 电源供给液晶屏，数字电源为其他控制器和 I/O 使用。在电源输入端进行滤波处理，保障电源的稳定性。

4.2.2　复位和启动电路

STM32L152 采用低电平复位，STM32L-Discovery 提供了一个阻容复位电路，用户可通过复位按键实现手动复位。上电时，复位引脚 NRST 为低，电容 C31 充电，充满后 NRST 被置高，完成复位。当用户按下复位键时，电容 C31 放电，NRST 被拉低，按键抬起后，电容充电，NRST 被拉高，完成复位。STM32L152 含内部复位电路，当 VDD 引脚电压小于 1.65V 时，芯片会保持在复位状态，如图 4-5 所示。

STM32L152 复位后开始执行程序，执行程序的位置与芯片的启动引脚 Boot0 和 Boot1 的状态有关，一般情况下，配置成 Flash 启动，即 Boot0 为低电平，复位后处理器从用户程序开始执行。在通过 ISP 下载程序的模式下，需要先运行片内 Flash 的 Bootloader 程序，该程序可以通过串口、USB 口、SPI、IIC 等方式将用户程序写入到用户 Flash，此时置 Boot1 为低电平，Boot0 为高电平。Boot1 和 Boot0 都为高电平时，复位后从 RAM 执行，由于掉电后 RAM 数据不保存，此模式一般只作为调试时使用，如图 4-6 所示。

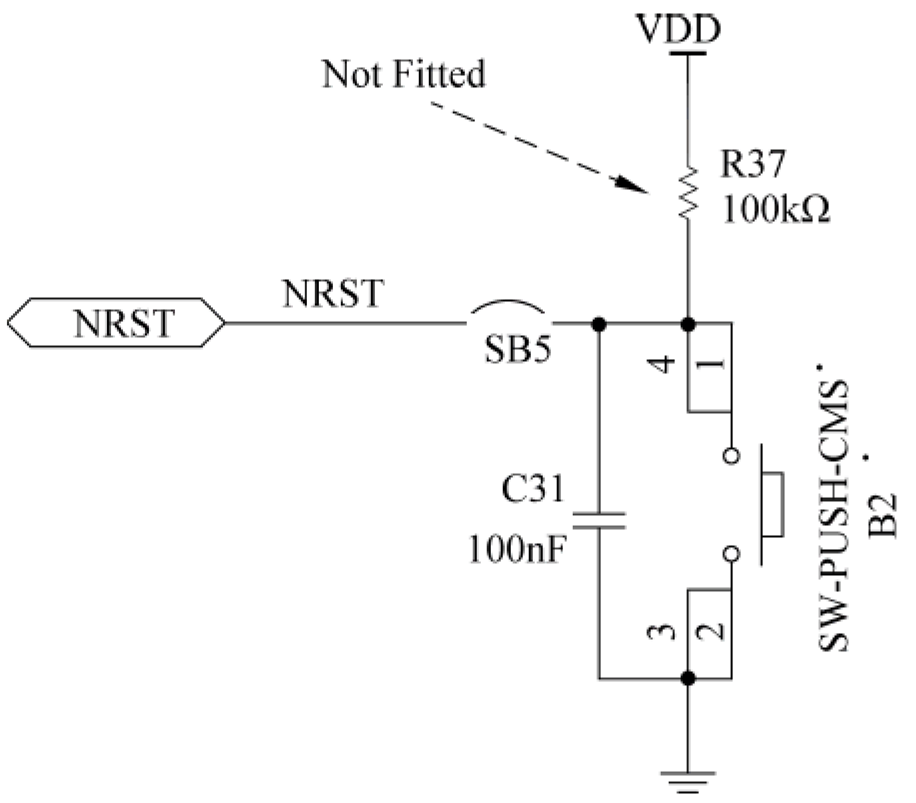


图 4-5　复位电路

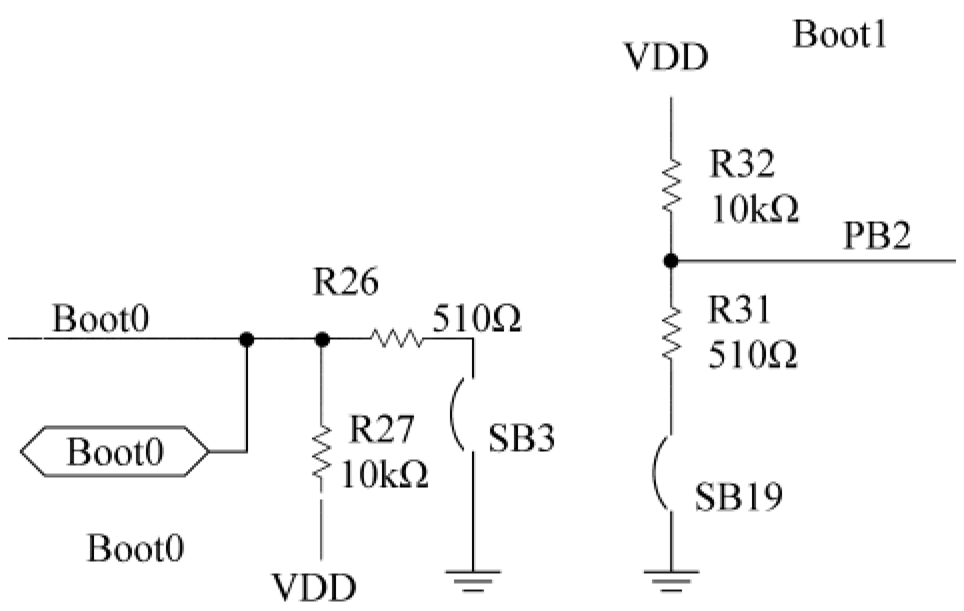


图 4-6　启动选择电路

STM32L-Discovery 开发板的 Boot0 默认置高，Boot1 由 PB2 引脚控制，电路板上电后，先执行 MCU 内部 Bootloader 程序，再跳转到用户代码去执行。Boot0 和 Boot1 的电平状态可以通过开发板硬件配置开关 SB3 和 SB19 更改。

4.2.3　时钟

STM32L152 支持多种时钟源，包括高速外部振荡器 HSE、低速外部振荡器 LSE、高速内部振荡器 HIS、多速率内部振荡器和低速内部振荡器 LSI。STM32L152 带有 16MHz 内

部 RC 振荡器 HSI，7 种频率的多速率内部 RC 振荡器 MSI，可以为 PLL 锁相环提供时钟，PLL 将时钟倍频到 32MHz 满速运行。37kHz 的低速 RC 振荡器 LSI 可用于实时时钟 RTC 以及看门狗 WDG。但由于内部 RC 振荡不精确，起振不稳定，因此一般会在开发板外置晶振。

外置晶振一般有两种：一种是高速外部晶振 HSE，可用作处理器的主时钟和外围控制器时钟，工作频率为 1M～24MHz；另一种是低速外部晶振 LSE，工作频率 32.768kHz，主要用于驱动实时时钟 RTC 和看门狗 WDG。外部晶振精度较高，而且频率越低精度越容易控制，因此一般 Timer，RTC 用 32.768kHz 的低速外部晶振。图 4-7 和图 4-8 是 STM32L-Discovery 开发板的外部时钟电路，高速外部晶振没有焊接，系统默认使用内部 RC 振荡器作为主时钟。

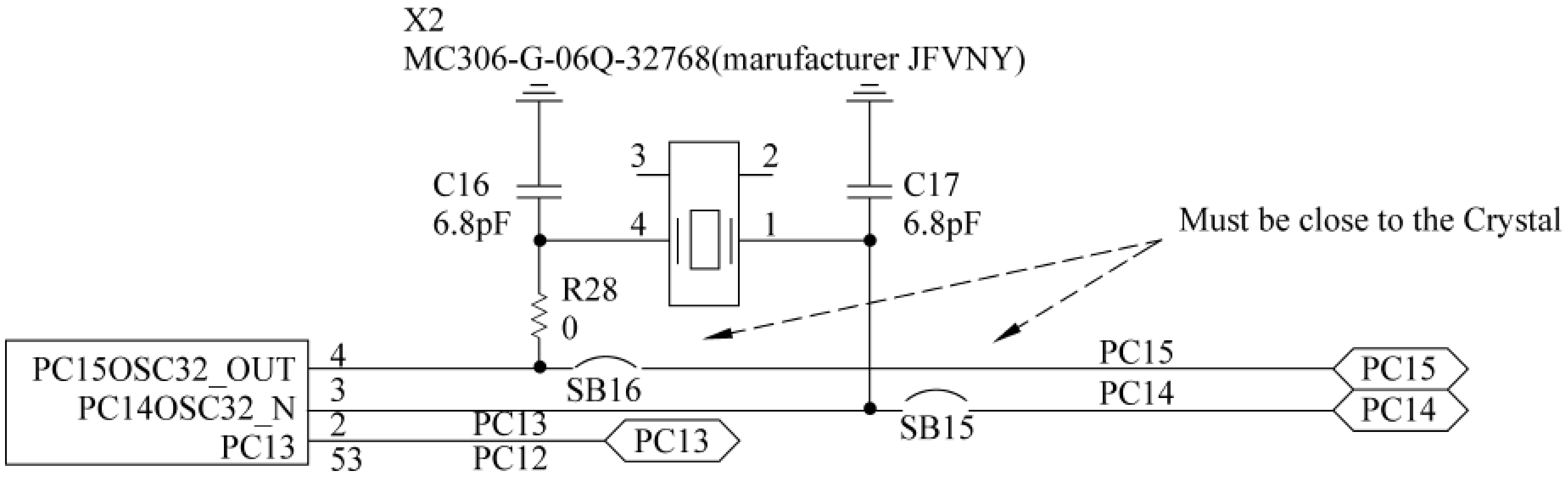


图 4-7 外部低速晶振电路

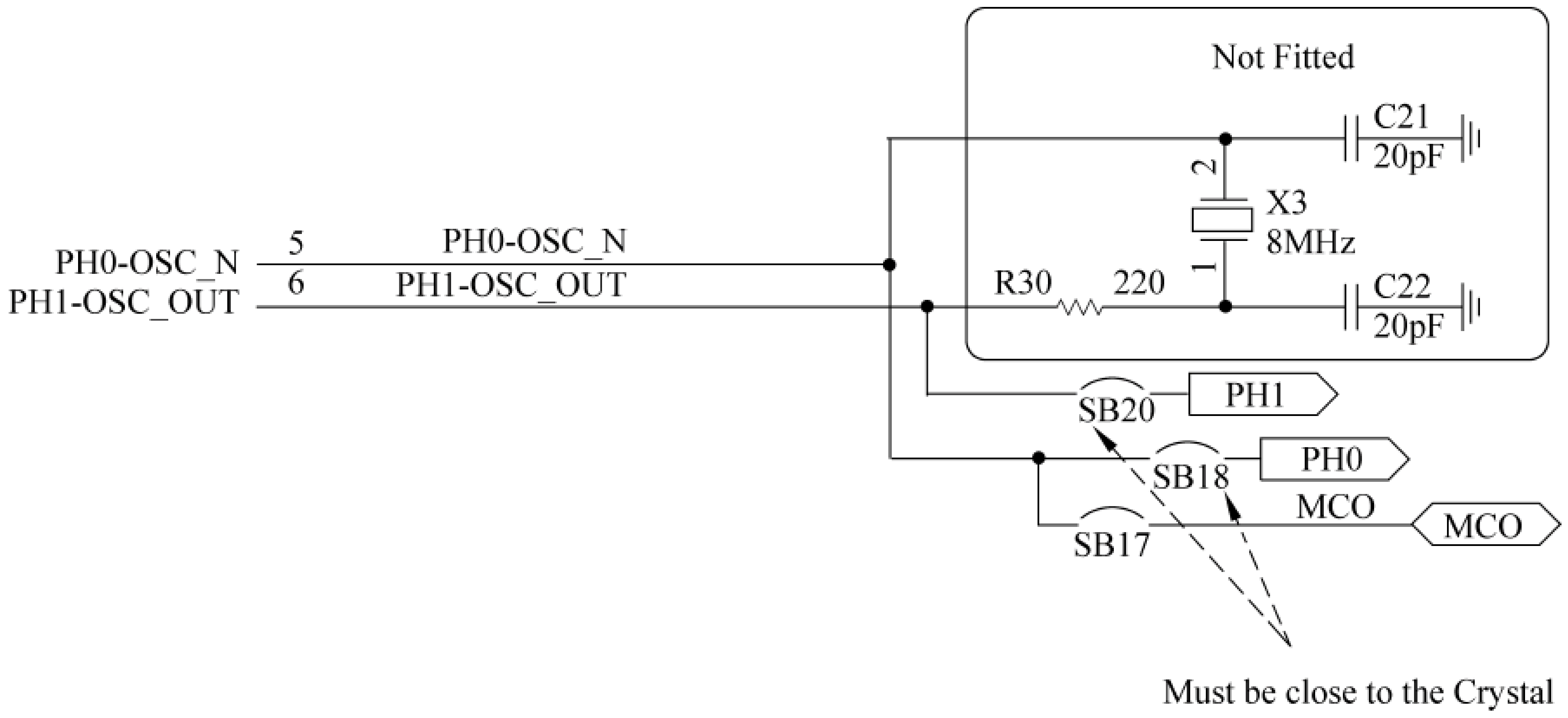


图 4-8 外部高速晶振电路

4.2.4 调试接口

STM32L152 采用 CoreSight 调试系统，内部没有 JTAG，而是调试访问接口 DAP，DAP 可以以 SWD 方式或 JTAG 的方式对外提供调试功能。SWD 只需要最少 2 根线

(SWCLK 和 SWDIO),JTAG 需要使用 5 根线。JTAG 接口和 SWD 接口共用,因此通过 JTAG 接口也可以采用 SWD 方式下载调试。STM32L-Discovery 板载一个 STLINK 调试器,该调试器与 STM32L152 处理器通过 SWD 方式连接,同时通过板子上的跳线可以单独使用该调试器连接其他处理器进行调试。图 4-9 为典型的 JTAG 调试接口电路,主要包含测试系统复位信号 nTRST、测试数据串行输入 TDI、测试模式选择 TMS、测试时钟 TCK 和测试数据串行输出 TDO,一般 JTAG 调试器为 20 口,将标准 JTAG 的 5 针进行了扩展。SWD 只使用 JTAG 中的 TCK 和 TMS,分别对应 SWCLK 和 SWDIO。目前大量 Cortex 系列处理器的调试器都支持 SWD 模式且占用的 I/O 口和面积小,因此推荐使用 SWD 模式。

4.2.5　按键

按键是系统设计中的主要输入源,STM32L-Discovery 开发板共有两个按键,其中一个用于系统复位,另一个为用户使用,其原理图如图 4-10 所示。按键连接在 PA0 口上,当按键按下时,PA0 为低电平,弹起时为高电平。该按键也可以用作 STM32L152 处理器唤醒输入或者外部中断输入。此外,STM32L-Discovery 开发板的触摸输入也可作为 4 个按键使用。

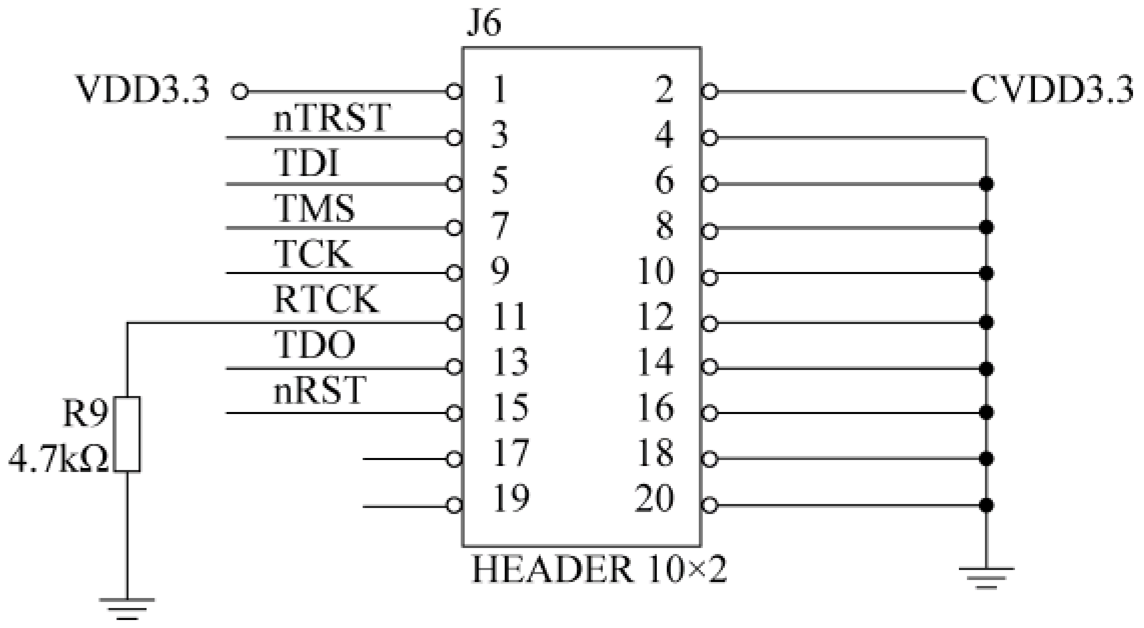


图 4-9　JTAG 调试接口电路

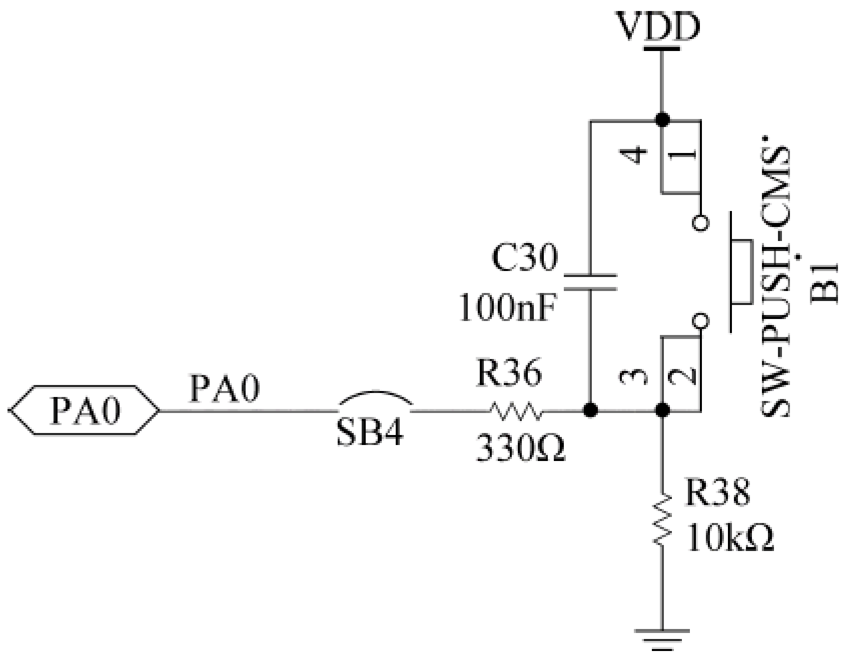


图 4-10　按键电路

4.2.6　LED 灯

STM32L-Discovery 开发板有 2 个用户 LED 灯,原理图如图 4-11 所示,LD3 和 LD4 分别连接在 PB7 和 PB6 上,当 PB7 输出为高电平时 LED3 变亮,反之变灭。此外,开发板提供了 2 个指示灯,其中 LD1 变绿色时指示 STLINK 调试器和处理器之间正在通信;LD2 为电源灯。

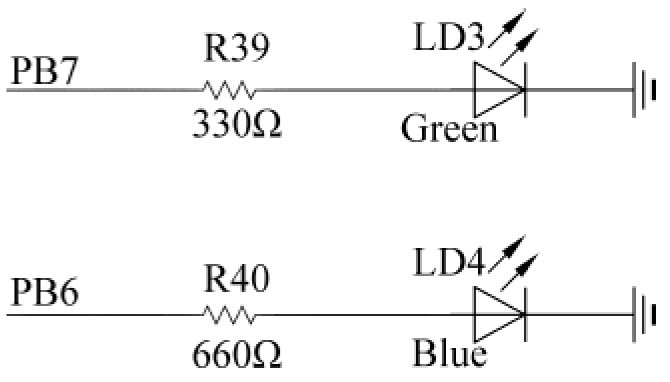


图 4-11　LED 灯电路

4.2.7 显示屏

液晶显示器(Liquid Crystal Display，LCD)是嵌入式系统常用的显示器件，由像素点或符号段组成，每个像素点或符号段是在两电极间放置液态的晶体，当电极间电压大于阈值电压时，控制杆状水晶分子改变方向，将光线折射出来变成可见。电极间的电压要交替变化以保护 LCD 不被损坏。

LCD 控制器的功能是产生显示驱动信号，驱动 LCD 显示器，用户只需要读写一系列的寄存器，完成配置和显示控制。STM32L152 内部集成 LCD 控制器，可以外接无源驱动的单色被动式 LCD 屏幕，最多可接 8 个 Common 端和 44 个 Segment 端的 320 像素 LCD。Common 端为水平方向，Segment 端为竖直方向，两者交叉即可控制像素的显示。

4.2.8 扩展 I/O 口

为便于扩展连接其他外设，STM32L152-Discovery 开发板将 CPU 的 I/O 引脚和电源连接到扩展插针上，如图 4-12 所示。其中包括 3V 和 5V 电源，以及除 PB0，PB1，PA6，PA7，PC4，PC5 之外的所有 I/O 引脚。

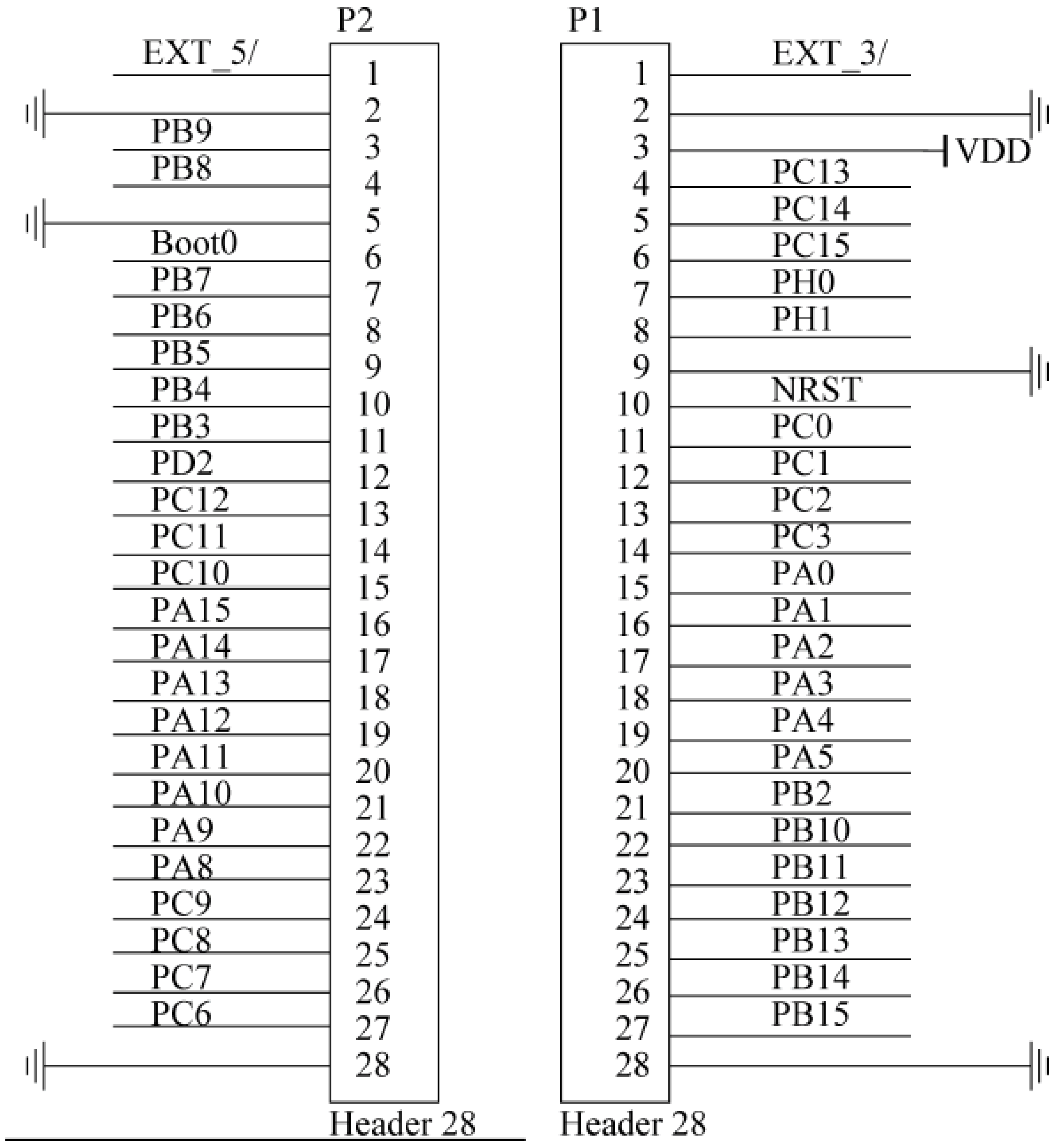


图 4-12 STM32L152-Discovery 扩展接口

扩展 I/O 引脚中，Boot0 已被置为高电平，NRST 为复位引脚，PA0 为按键和休眠唤醒

输入引脚，PA4 为电流采集输入引脚，PC13 为电流采集控制引脚，PA13，PA14，PB3 为 SWD 调试接口，PB6，PB7 为 LED 灯接口，PC14，PC15，PH0，PH1 分别为两个晶振引脚。其余的 33 个扩展 I/O 引脚中，除 PA5，PA11，PA12，PC12，PD2 外，被 LCD 占用。因此外接其他设备时，若需要使用特定功能的引脚或者多于 5 个普通 I/O 引脚，需要去除板子的 LCD 模块。

4.3　软件开发环境

4.3.1　嵌入式软件开发流程

嵌入式系统的软件开发与通用系统的软件开发有较大的区别，以下从交叉编译、交叉调试和固件(firmware)下载对嵌入式软件的编译、调试和固化进行介绍。

1. 交叉编译

程序首先要通过编译器将其转化为 CPU 可执行的机器代码，由于不同的处理器指令集不同，其所需的编译器也不同。在 PC 上开发程序，其编译的程序代码生成的是该 PC 的机器代码，也称为本地编译。

嵌入式软件开发采用交叉编译。交叉编译是指在一个平台上生成可以在另一个平台上可执行的代码的过程。由于目标嵌入式平台资源有限，无法或不便于进行程序的编辑、编译，因此我们在 PC 平台对目标嵌入式平台的程序进行编译，生成目标嵌入式平台的可执行代码。交叉编译的连接示例如图 4-13 所示。

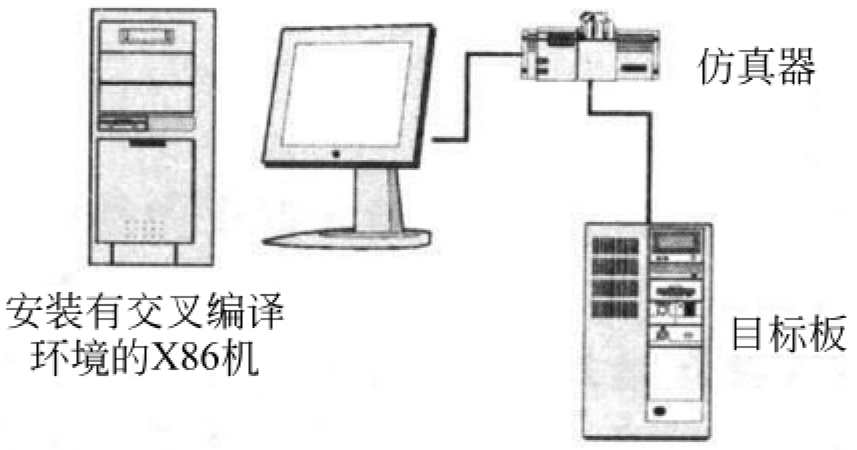


图 4-13　交叉编译环境

由于编译的过程包括编译和链接，因此，嵌入式的交叉编译也包括交叉编译、交叉链接，通常 ARM 的交叉编译器为 arm-elf-gcc、arm-linux-gcc 等，交叉链接器为 arm-elf-ld、arm-linux-ld 等。

2. 交叉调试

软件调试是软件开发过程中必不可少的一个环节，嵌入式软件的交叉调试与通用软件的调试方式有很大的差别。软件开发中，调试器与被调试的程序往往运行在同一台计算机上，调试器是一个单独运行着的进程，它通过操作系统提供的调试接口来控制被调试的进

程，实现单步、断点、变量查看等功能。而在嵌入式软件开发中，调试时采用的是在宿主机和目标机之间进行的交叉调试，调试器仍然运行在宿主机的操作系统之上，但被调试的程序却是运行在基于特定硬件平台的嵌入式操作系统中，调试器和被调试程序通过串口、网络或特殊硬件进行通信，调试器可以控制、访问被调试程序，读取或改变被调试程序的当前状态。

嵌入式系统的交叉调试主要分为软件方式和硬件方式两种。

1）软件方式

软件调试主要是通过插入调试桩的方式来进行的。调试桩方式进行调试是通过目标操作系统和调试器内分别加入某些功能模块，二者互通信息来进行调试。该方式的典型调试器有 gdb 调试器，一般只能用于调试运行于目标操作系统之上的应用程序，而不宜用来调试目标操作系统的内核代码及启动代码。

2）硬件调试

相对于软件调试而言，使用硬件调试器可以获得更强大的调试功能和更优秀的调试性能。硬件调试器的基本原理是通过仿真硬件的执行过程，让开发者在调试时可以随时了解到系统的当前执行情况。目前嵌入式系统开发中最常用到的硬件调试器有两种。

In-Circuit Emulator（ICE）方式：ICE 进行交叉调试时需要使用在线仿真器，它是目前最为有效的嵌入式系统的调试手段。它是仿照目标机上的 CPU 而专门设计的硬件，可以完全仿真处理器芯片的行为。仿真器与目标板可以通过仿真头连接，与宿主机可以通过串口、并口、网线或 USB 口等连接方式。由于仿真器自成体系，所以调试时既可以连接目标板，也可以不连接目标板。

在线仿真器提供了非常丰富的调试功能。在使用在线仿真器进行调试的过程中，可以按顺序单步执行，也可以倒退执行，还可以实时查看所有需要的数据，从而给调试过程带来了很多的便利。嵌入式系统应用的一个显著特点是与现实世界中的硬件直接相关，并存在各种异变和事先未知的变化，从而给微处理器的指令执行带来各种不确定因素，这种不确定性在目前情况下只有通过在线仿真器才有可能发现，但其价格比较昂贵，且不同的处理器需要不同的 ICE 硬件。

In-Circuit Debugger（ICD）方式：ICD 交叉调试时需要使用在线调试器，CPU 直接在其内部通过 JTAG 实现调试功能，并通过在开发板上引出的调试端口发送调试命令和接收调试信息，完成调试过程。JTAG 通过边界扫描技术实现对芯片输入输出信号的观察和控制。

3. 软件的固化与下载

小批量调试生产可通过调试器（JTAG 或 SWD 连接）进行 flash 的烧写固化，或者通过更改 BOOT 配置，利用 MCU 中内嵌的 bootloader 程序进行串行 flash 烧写，当需要大批量生产时，可采用专用的 flash 烧写器。

由于 ARM 芯片的广泛使用，众多开发工具都支持 ARM 平台程序的开发，ARM 开发的编译器主要有 ARMCC 和 ARM-LINUX-GCC 两种，前者是 ARM 公司出品的编译器，完全符合 ARM 指令集格式，后者是基于 Linux 的 ARM 交叉编译器，使用的是 GNU 的汇编方式，除了指令集与 ARM 指令兼容外，还支持一些非 ARM 标准的语法。由于开发软件越来越复杂，因此集成开发环境整合了编辑器、汇编器、编译器、调试器、模拟器等功能，使得应

用系统的开发更为便捷。

ARM 的集成开发环境主要有 ADS，KEIL MDK 和 IAR EWARM 等。ADS 是 ARM 公司早期的集成开发环境，采用 CodeWarrior IDE，并集成了 ARM 开发包和应用库。KEIL 原本是单片机开发环境，被 ARM 收购后作为 ADS 的升级版本 MDK 推出，被广泛使用。第三方开发的集成开发环境中，IAR EWARM 的编译效率和优化表现最为突出。

4.3.2　程序开发库 CMSIS

在程序设计中，我们会大量使用到库函数，为了便于 ARM Cortex 系列处理器的程序设计，ARM 定义了 ARM Cortex 微控制器软件接口标准 CMSIS。CMSIS 为开发者访问底层硬件提供了一个 API 接口，通过使用固件函数库，无需深入掌握底层硬件细节就可以对外设进行控制。CMSIS 由 ARM 和芯片厂家提供，ARM 提供独立于芯片的内核设备访问层、中间设备访问的通用方法以及外设访问接口，芯片厂家在 CMSIS 基础上提供针对自己芯片的外设接口库，包含了 GPIO、TIMER、CAN、I2C、SPI、UART 和 ADC 等所有标准外设，便于进行二次开发和应用，可以大大减少用户的程序编写时间，进而降低开发成本，缩短在不同处理器之间的移植时间。

CMSIS 为 Cortex-M 微控制器系统定义了：

- 访问外设寄存器的通用方法和定义异常向量的通用方法。
- 内核设备的寄存器名称和内核异常向量的名称。
- 独立于微控制器的 RTOS 接口，带调试通道。
- 中间设备组件接口（TCP/IP 协议栈，闪存文件系统）。

如图 4-14 所示，CMSIS 由 4 部分组成：

- CMSIS-CORE 组件：提供 Cortex-M 系列微处理器的内核和外设的寄存器定义、中断接口和 API 函数，提供系统启动方法和访问特定处理器功能和内核外设的函数。

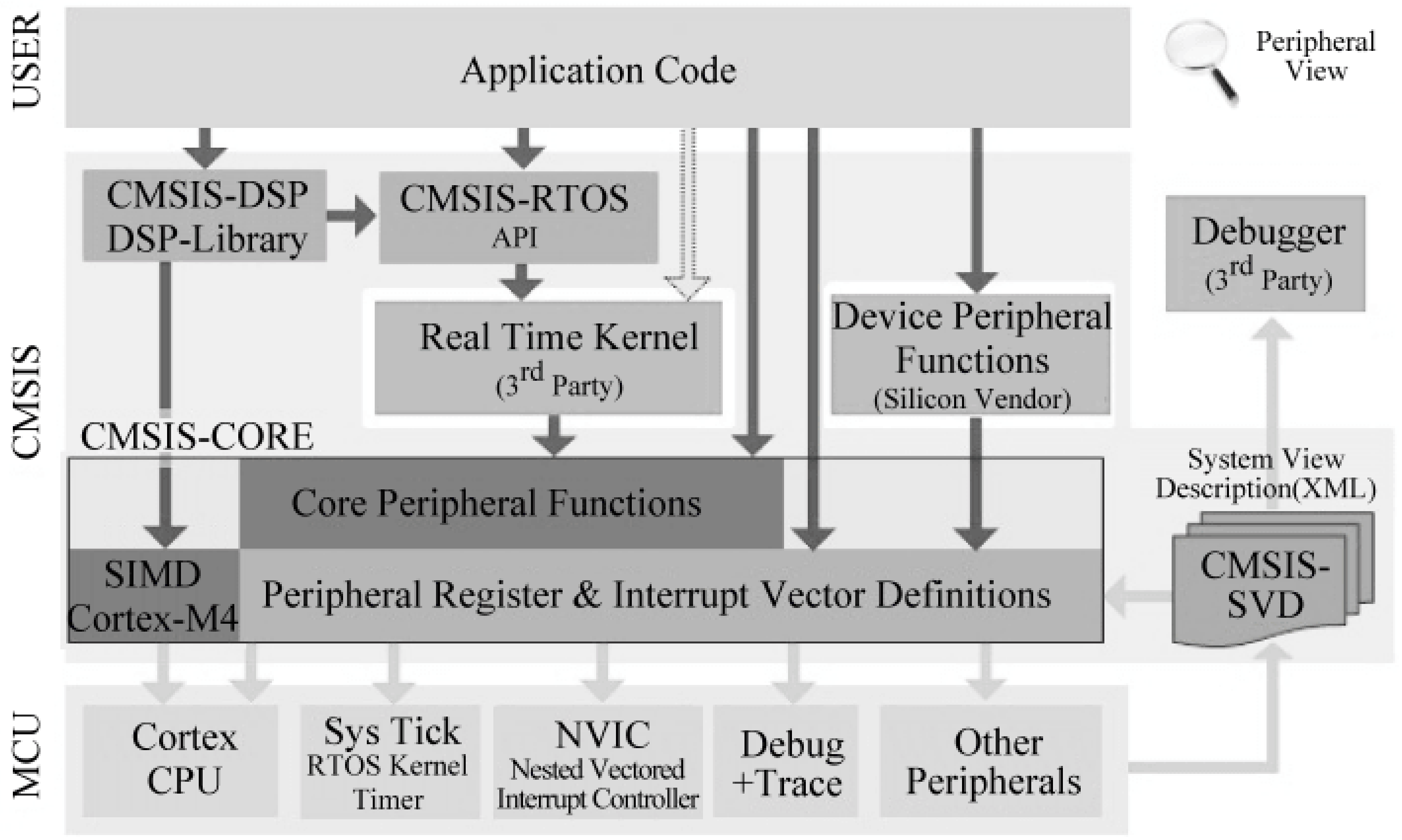


图 4-14　CMSIS 库的作用

- CMSIS-DSP 组件：优化的信号处理算法。
- CMSIS-SVD 组件：描述设备外设和中断的 XML 文件。
- CMSIS-RTOS API 组件：提供通用的 API 接口给一些实时操作系统(RTOS)。

CMSIS 提供的独立于编译器的主要内核文件包括：

- Cortex-M3 内核及其设备文件(core_cm3. h ＋ core_cm3. c)：用于访问 Cortex-M3 内核及其 NVIC,SysTick 等设备,访问 Cortex-M3 CPU 寄存器和内核外设的函数。
- 微控制器专用头文件(device. h)：用于指定中断号码(与启动文件一致),定义外设寄存器 (寄存器的基地址和布局)。
- 微控制器专用系统文件(system_device. c)：包括 SystemInit(),SystemFrequncy(), Sysem_ExtMemCtl()等函数,用来初始化处理器。

CMSIS 提供的与编译器相关的启动文件主要是编译器启动代码 (startup_device. s),它定义了中断处理程序列表及中断处理程序默认函数。

基于 CMSIS,ST 提供了 STM32Lx 系列的内核设备访问 API 和外围设备访问 API,可在 http：//www. st. com/web/en/catalog/tools/PF257908 下载,该库提供了 STM32L 系列中高、中、低密度处理器的编译器相关启动代码,内核设备文件、MCU 系统文件,以及外围控制器的所有驱动程序。

STM32Lx 系列的 CMSIS 库的结构如图 4-15 所示,CMSIS/Device/ST/STM32Lxx 目

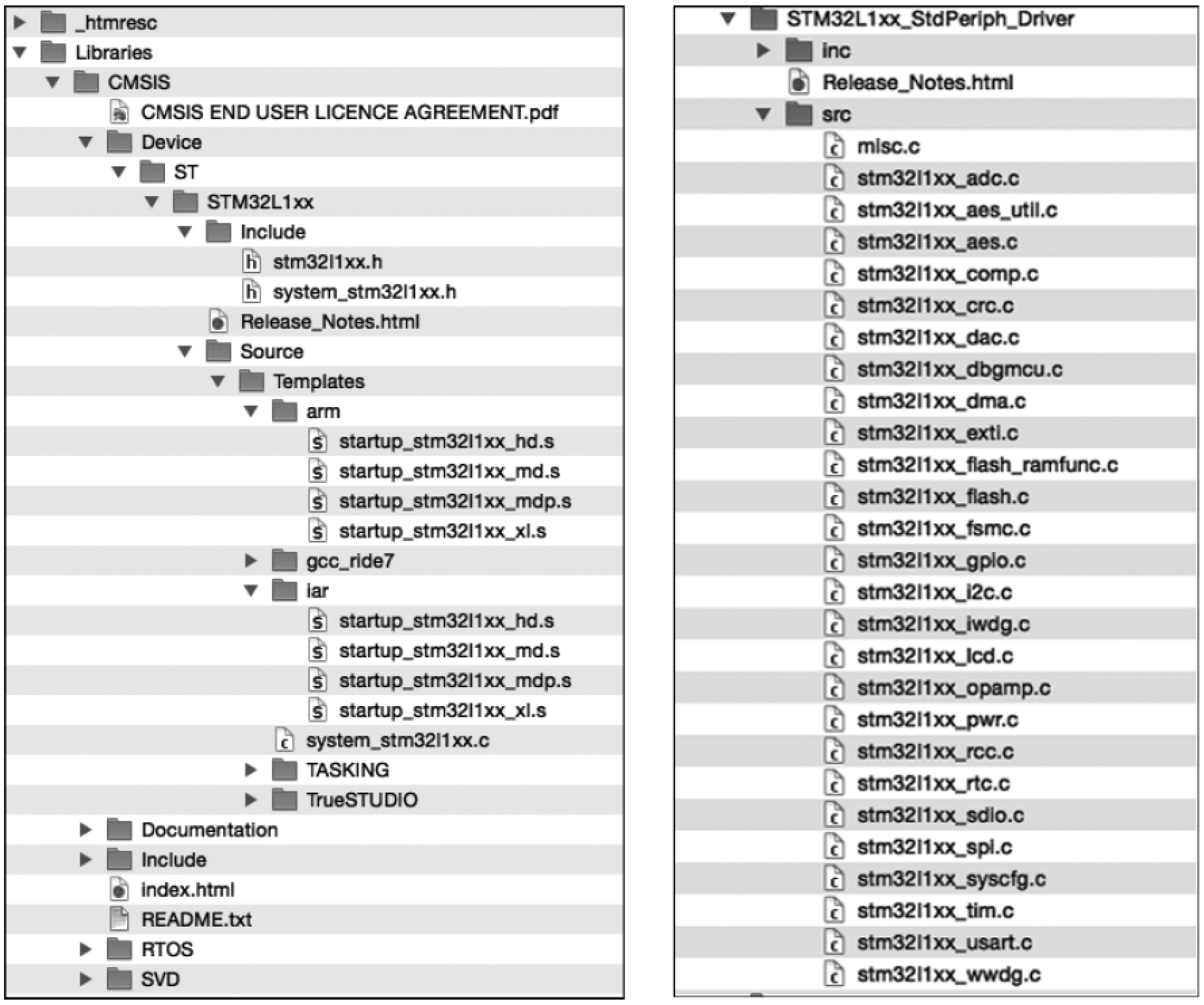


图 4-15　CMSIS 代码树

录下的 include 文件夹包含微控制器专用头文件 stm32l1xx.h 和微控制器专用系统文件的头文件 system_stm32l1xx.h;source/Templates 文件夹下包含的是微控制器专用系统文件 system_stm32l1xx.c 和编译器相关启动文件，其中 ARM 目录下是 ARMCC 编译器(KEILMDK 开发环境)的启动文件;IAR 目录下是 IAR EWARM 开发环境的启动文件，根据微控制器 flash 容量大小的不同，选用不同的启动文件。STM32L152-Discovery 开发板使用的 STM321L152RBT6 属于中密度型，因此在构建工程时选择 startup_stm32l1xx_md.s。

STM32L1xx_StdPeriph_Driver /src 目录下为 STM32L1xx 系列微控制器的外设驱动程序，包括 ADC、AES、Flash、LCD、GPIO 等。用户利用 ST 提供的 CMSIS 库可以方便地调用这些驱动程序实现快速开发。

此外，为便于开发，ST 还提供了 STM32LCube 库，实现了网络协议栈、USB 设备操作、图形化界面、文件系统、实时操作系统等接口。

4.3.3　STM32L52 嵌入式程序开发预备知识

1. C 语言的位操作

位操作是嵌入式系统中对于外围控制器寄存器操作的重要方法，本节对 C 语言中的位操作进行概要介绍。

C 语言支持 6 种位操作：

(1) &：按位“与”，对两个操作数的每一位进行逻辑与操作，例如：

```
1000 1000 & 1000 0001 = 1000 0000;
```

(2) |：按位“或”，对两个操作数中的每一位进行逻辑或操作，例如：

```
1000 1000 | 1000 0001 = 1000 1001;
```

(3) ^：按位“异或”，对两个操作数中的每一位进行逻辑异或操作，仅当两个操作数不同时，相应的输出结果才为 1，否则为 0，例如：

```
1000 1000 ^ 1000 0001 = 0000 1001;
```

(4) ～：按位“取反”，将操作数中地每一位取反，例如：

```
~1000 1000 = 0111 0111;
```

(5) <<：“算术左移”操作，将操作数的各位按要求向左移动若干位，例如 5<<3 等价于 00000101<<3，即 0010 1000。算术左移相当于乘法，左移 n 位的结果等于原操作数乘 2 的 n 次方。

(6) >>：“算术右移”，将操作数的各位按要求向右移动若干位，例如 4>>2 等价于 0000 0100>>2，即 0000 0001。算术右移相当于除法，右移 n 位的结果等于原操作数除 2 的 n 次方。

位运算符主要用于快速乘除运算和寄存器操作。

(1) 快速乘除运算：移位操作可用于整数的快速乘除运算，左移一位等效于乘 2，而右移一位等效于除以 2。

如：x = 7，二进制表达为：0000 0111，

```
x<<1            0000 1110,相当于:x=2*7=14,
x<<3            0111 0000,相当于:x=14*2*2*2=112
x<<2            1100 0000,相当于:x=112*2*2=448-256=192
```

在作第三次左移时，其中一位为 1 的位移到外面去了，而左边只能以 0 补齐，因而便不等于 112 * 2 * 2=448，而是等于 192 了。当 x 按刚才的步骤反向移动回去时，就不能返回到原来的值了，因为左边丢掉的一个 1，再也不能找回来了：

```
x>>2            0011 0000,相当于 x=192/4=48
x>>3            0000 0110,相当于 x=48/8=6
x>>1            0000 0011  相当于 x=6/2=3
```

(2) 寄存器位操作：寄存器置指定位置为 1：PORTA |= (1<<n)，PORTA 的第 n 为置为 1，其他位不变。例如：

```
PORTA |= (1<<4) :将第四位置 1:
PORTA |= (1<<7) | (1<<4) | (1<<0) 将设第 7、4 和 0 位置 1
```

将寄存器指定位置为 0：PORTA &= ~(1<<n)，寄存器的第 n 位将被清 0，但不影响其他位，例如：

```
PORTA &=~(1<<4):第四位置 0
```

2. 用 C 语言操作硬件

第 3 章 3.3 节的启动代码分析中，我们已经可以实现从 MCU 上电复位到跳转到 C 语言的 main 函数开始执行，这样我们可以使用 C 语言对硬件进行控制。对于 C 程序，程序编译时已经指定了 Flash 的地址和 SRAM 的地址，因此建立堆栈后，数据处理类的程序即可正常运行。但通常我们要操作外设，配置外设的寄存器控制外设运行，从第 2 章的存储空间映射可知外设空间和 Flash、SRAM 统一遍址，对外设寄存器的访问就是对存储空间的访问。

Cortex-M3 的 SysTick 定时器的当前值寄存器地址为 0xE000E018，用 C 语言访问这个地址的一个字，即可读写该寄存器的值。

```
unsigned int *p=(unsigned int *)(0xE000E018)
unsigned int SysTick_Value=*p;          //读取 SysTick 计数器值
*p=SysTick_Value+2000;                  //向 SysTick 计数器写入新值
```

利用宏定义，可以给每个寄存器起一个名字。

```
#define SYSTICK_VALUE_R   (*(unsigned int*)0xE000E018)
```

这样可以直接使用寄存器名：

```
SYSTICK_VALUE_R = 20000;
```

通常我们将寄存器定义为如下形式：

```
#define SYSTICK_VALUE_R   (*(volatile unsigned int *)0xE000E018)
```

volatile 是 C 的一个关键字，别称易失变量，即容易丢失的变量；因为编译器为了程序的效率，在编译时会进行一些优化。在变量前加上个 volatile 关键字，编译器就不会对该变量进行优化了，这样可以保证读取的是存储器地址而不是缓存或寄存器（优化后可能把该变量的值存放在某个临时的寄存器中，这样会导致寄存器和存储器内容不一致）。

再结合 C 语言的位操作运算，就可以对寄存器的任意 bit 进行访问和控制。CMSIS 提供了寄存器的规范定义和 HAL 硬件抽象层的 C 库版本。

3. 时钟树及时钟配置

STM32L152 的时钟树如图 2-17 所示。主要时钟源有以下 5 个：

1）HSE 时钟

高速外部时钟信号（HSE）可以由 HSE 外部晶体/陶瓷谐振器和 HSE 用户外部时钟产生。为了减少时钟输出的失真和缩短启动稳定时间，晶体/陶瓷谐振器和负载电容器要尽可能地靠近振荡器引脚。负载电容值必须根据所选择的振荡器来调整。外部晶体/陶瓷谐振器可以提供一个非常精确的时钟源，HSE 晶体可以通过设置时钟控制寄存器里 RCC_CR 中的 HSEON 位被启动和关闭。

2）HSI 时钟

HSI 时钟信号由内部 16MHz 的 RC 振荡器产生，可直接作为系统时钟或作为 PLL 输入。HSI RC 振荡器能够在不需要任何外部器件的条件下提供系统时钟。它的启动时间比 HSE 晶体振荡器短。然而，即使在校准之后它的时钟频率精度仍较差。HSI RC 可由时钟控制寄存器中的 HSION 位来启动和关闭。

3）LSE 时钟

LSE 晶体是一个 32.768kHz 的低速外部晶体或陶瓷谐振器。它为实时时钟或者其他定时功能提供一个低功耗的精确时钟源。LSE 晶体通过在备份域控制寄存器（RCC_BDCR）里的 LSEON 位启动和关闭。

4）LSI 时钟

LSI RC 担当一个低功耗时钟源的角色，它可以在停机和待机模式下保持运行，为独立看门狗和自动唤醒单元提供时钟。LSI 时钟频率为 37kHz。LSI RC 可以通过控制/状态寄存器（RCC_CSR）里的 LSION 位来启动或关闭。

5）MSI 时钟

MSI 是内部集成的多速率时钟，有 65.5kHz、131kHz、262kHz、524kHz、1.05MHz、2.1MHz 和 4.2MHz 7 种配置。

输入时钟源的频率一般都比较低，系统所需的时钟频率可能会比较高，因此我们可以采用 PLL 锁相环对输入时钟进行倍频处理。内部 PLL 可以用来倍频 HSI 的 RC 振荡时钟或

HSE 晶体时钟。PLL 的设置必须在其被激活前完成。一旦 PLL 被激活，这些参数就不能被改动。在使用 USB 时，PLL 必须被设置为输出 48 MHz 时钟提供 USBCLK 时钟。

对于微控制器系统，所需的时钟包括系统主时钟 SYSCLK，用于 Cortex-M3 处理器核心；AHB 总线时钟 HCKL，用于 AHB 总线和连接到 AHB 的外设；APB1 总线时钟 PPB1 和 APB2 总线时钟 PPB2，用于外围总线和连接到外围总线的外设。此外 USB、RTC 需要特殊的时钟需要配置，微控制器还提供一个输出 MCO，可以将 MCU 内部的时钟输出到芯片引脚上。

系统复位后，2.1MHz 的 MSI 被选为系统时钟。当时钟源被直接或通过 PLL 间接作为系统时钟时，它将不能被停止。只有当目标时钟源准备就绪时才可以进行系统时钟切换。时钟控制寄存器(RCC_CR)里的状态位指示哪个时钟已经准备好了，哪个时钟目前被用作系统时钟。

4. 时钟和复位配置控制器

1）时钟控制寄存器 RCC_CR

时钟控制寄存器用于配置内部和外部的高速时钟以及锁相环 PLL 的开启控制，其有效域定义如图 4-16 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	RTCPRE[1:0]		CSS ON	Reserved		PLL RDY	PLLON	Reserved					HSE BYP	HSE RDY	HSE ON
	rw	rw	rw			r	rw						rw	r	rw
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Reserved						MSI RDY	MSION	Reserved						HSI RDY	HSION
						r	rw							r	rw

图 4-16　时钟控制寄存器

HSION、MSION、HSEON、PLLON 分别用于控制 HSI、MSI、HSE 和 PLL 是否启用，若启用其中一个时钟源，其稳定后，对应的 HSIRDY、HSERDY、PLLDRY 和 MSIRDY 会自动置 1，表明该时钟可用。

该寄存器的初值为 0b0XX0 0000 0000 0X00 0000 0011 0000 0000，即默认 MSION 开启，MCU 上电后采用 MSI 时钟。

2）内部时钟源及时钟校准寄存器 RCC_ICSCR

内部时钟源及时钟校准寄存器的有效域定义如图 4-17 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MSITRIM[7:0]								MSICAL[7:0]							
rw	rw	rw	rw	rw	rw	rw	rw	r	r	r	r	r	r	r	r
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
MSIRANGE[2:0]			HSITRIM[4:0]					HSICAL[7:0]							
rw	rw	rw	rw	rw	rw	rw	rw	r	r	r	r	r	r	r	r

图 4-17　内部时钟源及时钟校准寄存器

MCU 上电后采用 MSI 时钟，MSI 为多速率时钟，RCC_ICSCR 寄存器的 MSIRANGE 域用于指定 MSI 时钟的频率，000 表示 65.536kHz，001 表示 131.072kHz，010 表示

262.144kHz,011 表示 524.288kHz,100 表示 1.048MHz,101 表示 2.097MHz(默认值),110 表示 4.194MHz,111 无效。该寄存器默认值为 0x00XX B0XX,即 MSIRANGE 配置为 101,MSI 频率 2.097MHz。

3) 时钟配置寄存器 RCC_CFGR

时钟配置寄存器用于配置 PLL,AHB、APB 的总线时钟,其有效域定义如图 4-18 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	MCOPRE[2:0]			Res.	MCOSEL[2:0]			PLLDIV[1:0]		PLLMUL[3:0]				Res.	PLL SRC
	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw	rw	rw		rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved		PPRE2[2:0]			PPRE1[2:0]			HPRE[3:0]				SWS[1:0]		SW[1:0]	
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	r	r	rw	rw

图 4-18　时钟配置寄存器

MCOPRE[2:0],MCU 输出时钟 MCO 的分频因子,000~100 分别表示 1、2、4、8、16 分频,其余配置无效。

MCOSEL[2:0]用于配置 MCO 输出时钟的时钟源,000 表示 MCO 输出禁用。001 表示 SYSCLK 作为 MCO 的输出源,010 表示 HSI 作为 MCO 的输出源,011 表示 MSI 作为 MCO 的输出源,100 表示 HSE 作为 MCO 的输出源,101 表示 PLL 作为 MCO 的输出源,110 表示 LSI 作为 MCO 的输出源,111 表示 LSE 作为 MCO 的输出源。

PLLDIV[1:0],PLL 时钟的分频系数,00 表示不分频,01~11 分别表示 2、3、4 分频。

PLLMUL[3:0]:PLL 的倍频系数,0000~1000 分别表示 3、4、6、8、12、16、24、32、48 倍频,其余配置无效。

PLLSRC,PLL 的输入时钟选择,0 表示 HIS,1 表示 HSE。

PPRE2[2:0]和 PPRE1[2:0]分别用于配置 APB2、APB1 的总线时钟分频系数,APB 的时钟来源于 AHB 的 HCLK,0xx 表示 HCLK 不分频,100~111 分别表示 2、4、8、16 分频。

HPRE[3:0],AHB 时钟分频系数,AHB 的时钟来源于 SYSCLK,0xxx 表示 SYSCLK 不分频,1000~1111 分别表示 2、4、8、16、64、128、256 和 512 分频。

SW[1:0],SYSCLK 来源配置,00 表示 MSI 作为系统时钟,01 示 HSI 作为系统时钟,10 表示 HSE 作为系统时钟,11 表示 PLL 作为系统时钟。

SWS[1:0],系统时钟的状态,只读,用于表示那个时钟源正在被作为系统时钟,00~11 分别表示 MSI、HSI、HSE 和 PLL。

4) AHB 外围时钟使能寄存器 RCC_AHBENR

AHB 外围时钟使能寄存器用于配置连接到 AHB 总线上的每个外设时钟是否启用,其有效域定义如图 4-19 所示,写 1 表示启用,写 0 表示关闭时钟。

5) APB2 外围时钟使能寄存器 RCC_APB2ENR

APB2 为时钟使能寄存器,用于配置连接到 APB2 总线上的每个外设时钟是否启用,其有效域定义如图 4-20 所示,写 1 表示启用,写 0 表示关闭时钟。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	FSMC EN	Reserved		AES EN	Res.	DMA2EN	DMA1EN	Reserved							
	rw			rw		rw	rw								
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
FLITF EN	Reserved		CRCEN	Reserved				GPIOG EN	GPIOF EN	GPIOH EN	GPIOE EN	GPIOD EN	GPIOC EN	GPIOB EN	GPIOA EN
rw			rw					rw	rw	rw	rw	rw	rw	rw	rw

图 4-19　AHB 外围时钟使能寄存器

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Res.	USART1 EN	Res.	SPI1 EN	SDIO EN	Res.	ADC1 EN	Reserved				TIM11 EN	TIM10 EN	TIM9 EN	Res.	SYSCF GEN
	rw		rw	rw		rw					rw	rw	rw		rw

图 4-20　APB2 外围时钟使能寄存器

6）APB1 外围时钟使能寄存器 RCC_APB1ENR

APB1 时钟使能寄存器用于配置连接到 APB1 总线上的每个外设时钟是否启用，其有效域定义如图 4-21 所示，写 1 表示启用，写 0 表示关闭时钟。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
COMP EN	Res.	DAC EN	PWR EN	Reserved				USB EN	I2C2 EN	I2C1 EN	UART5 EN	UART4 EN	USART3 EN	USART2 EN	Res.
rw		rw	rw					rw	rw	rw	rw	rw	rw	rw	
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
SPI3 EN	SPI2 EN	Reserved		WWD GEN	Res.	LCD EN	Reserved			TIM7 EN	TIM6 EN	TIM5 EN	TIM4 EN	TIM3 EN	TIM2 EN
rw	rw			rw		rw				rw	rw	rw	rw	rw	rw

图 4-21　APB1 外围时钟使能寄存器

7）控制/状态寄存器 RCC_CSR

控制和状态寄存器的有效域定义如图 4-22 所示。RCC_CSR 中涉及 LSI 和 LSE 的启用和配置，分别用 LSION、LSEON、LSIRDY、LSERDY 表示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
LPWR RSTF	WWDG RSTF	IWDG RSTF	SFT RSTF	POR RSTF	PIN RSTF	OBLRS TF	RMVF	RTC RST	RTC EN	Reserved				RTCSEL[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw					rw	rw
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Reserved			LSECS SD	LSECS SON	LSE BYP	LSERDY	LSEON	Reserved						LSI RDY	LSION
			r	rw	rw	r	rw							r	rw

图 4-22　控制/状态寄存器

8）时钟配置的相关寄存器和库函数

（1）寄存器结构定义

CMSIS 中 RCC 寄存器结构定义为 RCC_TypeDeff，在文件“stm32L1xx. h”中定义

如下：

```
typedef struct
{
    __IO uint32_t CR;                        //时钟控制寄存器
    __IO uint32_t CFGR;                      //时钟配置寄存器
    __IO uint32_t CIR;                       //时钟中断寄存器
    __IO uint32_t AHBRSTR                    //AHB 外设复位寄存器
    __IO uint32_t APB2RSTR;                  //APB2 外设复位寄存器
    __IO uint32_t APB1RSTR;                  //APB1 外设复位寄存器
    __IO uint32_t AHBENR;                    //AHB 外设时钟使能寄存器
    __IO uint32_t APB2ENR;                   //APB2 外设时钟使能寄存器
    __IO uint32_t APB1ENR;                   //APB1 外设时钟使能寄存器
    __IO uint32_t AHBLPENR                   //AHB 低功耗模式使能寄存器
    __IO uint32_t   APB2LPENR                //APB2 低功耗模式使能寄存器
    __IO uint32_t APB1LPENR                  //APB1 低功耗模式使能寄存器
    __IO uint32_t CSR;                       //控制/状态寄存器
} RCC_TypeDef;
```

RCC 外设的寄存器地址定义在文件“stmL1xx.h”：

```
#define PERIPH_BASE ((uint32_t)0x40000000)
#define APB1PERIPH_BASE PERIPH_BASE
#define APB2PERIPH_BASE (PERIPH_BASE + 0x10000)
#define AHBPERIPH_BASE (PERIPH_BASE + 0x20000)
#define RCC_BASE (AHBPERIPH_BASE + 0x1000)
#define RCC ((RCC_TypeDef * ) RCC_BASE)
```

（2）配置案例

```
static void SetSysClock(void)
{
    __IO uint32_t StartUpCounter = 0, HSEStatus = 0;
    RCC->CR |= ((uint32_t)RCC_CR_HSEON);                //使能 HSE
    do                                                  //等待 HSE 工作或超时
    {
        HSEStatus = RCC->CR & RCC_CR_HSERDY;
        StartUpCounter++;
    } while((HSEStatus == 0) && (StartUpCounter != HSE_STARTUP_TIMEOUT));
    if ((RCC->CR & RCC_CR_HSERDY) != RESET)
        HSEStatus = (uint32_t)0x01;
    else
        HSEStatus = (uint32_t)0x00;
    if (HSEStatus == (uint32_t)0x01)                    //HSE 正常工作
    {
```

```
        //配置 CFG 寄存器,HCLK、APB1、APB2 分频系数均为 1
        RCC->CFGR |= (uint32_t)RCC_CFGR_HPRE_DIV1;                  //HCLK = SYSCLK/1
        RCC->CFGR |= (uint32_t)RCC_CFGR_PPRE2_DIV1;                 //PCLK2 = HCLK/1
        RCC->CFGR |= (uint32_t)RCC_CFGR_PPRE1_DIV1;                 //PCLK1 = HCLK/1
        //清除 PLL 控制位,配置 PLL 的时钟源、倍频系数、分频系数
        RCC->CFGR &= (uint32_t)((uint32_t)~(RCC_CFGR_PLLSRC | RCC_CFGR_PLLMUL |
                                      RCC_CFGR_PLLDIV));
        RCC->CFGR |= (uint32_t)(RCC_CFGR_PLLSRC_HSE | RCC_CFGR_PLLMUL12 |
                             RCC_CFGR_PLLDIV3);
        RCC->CR |= RCC_CR_PLLON;                                    //启用 PLL
        while((RCC->CR & RCC_CR_PLLRDY) == 0);                      //等待 PLL 稳定
        //配置 CFGR 的 SW 选择 PLL 时钟作为系统时钟源
        RCC->CFGR &= (uint32_t)((uint32_t)~(RCC_CFGR_SW));
        RCC->CFGR |= (uint32_t)RCC_CFGR_SW_PLL;
        //读 CFGR 的 SWS,等待系统时钟状态指示 PLL 时钟成为系统时钟
        while ((RCC->CFGR&(uint32_t)RCC_CFGR_SWS) != (uint32_t)RCC_CFGR_SWS_PLL);
    }
    else
        //HSE 不工作,使用原有时钟即可
}
```

(3) RCC 库函数

RCC 库函数如表 4-1 所示。

表 4-1 为 ST 提供的 RCC 控制相关库函数

函 数 名	功 能
RCC_DeInit	将外设 RCC 寄存器重设为默认值
RCC_HSEConfig	设置外部高速晶振(HSE)
RCC_WaitForHSEStartUp	等待 HSE 起振
RCC_HSICmd	使能或者失能内部高速晶振(HSI)
RCC_PLLConfig	设置 PLL 时钟源及倍频系数
RCC_PLLCmd	使能或者失能 PLL
RCC_SYSCLKConfig	设置系统时钟(SYSCLK)
RCC_GetSYSCLKSource	返回用作系统时钟的时钟源
RCC_HCLKConfig	设置 AHB 时钟(HCLK)
RCC_PCLK1Config	设置低速 AHB 时钟(PCLK1)
RCC_PCLK2Config	设置高速 AHB 时钟(PCLK2)
RCC_ITConfig	使能或者失能指定的 RCC 中断

续表

函　数　名	功　　能
RCC_ADCCLKConfig	设置 ADC 时钟(ADCCLK)
RCC_LSEConfig	设置外部低速晶振(LSE)
RCC_LSICmd	使能或者失能内部低速晶振(LSI)
RCC_RTCCLKConfig	设置 RTC 时钟(RTCCLK)
RCC_RTCCLKCmd	使能或者失能 RTC 时钟
RCC_GetClocksFreq	返回不同片上时钟的频率
RCC_AHBPeriphClockCmd	使能或者失能 AHB 外设时钟
RCC_APB2PeriphClockCmd	使能或者失能 APB2 外设时钟
RCC_APB1PeriphClockCmd	使能或者失能 APB1 外设时钟
RCC_GetFlagStatus	检查指定的 RCC 标志位设置与否
RCC_GetITStatus	检查指定的 RCC 中断发生与否
RCC_ClearITPendingBit	清除 RCC 的中断待处理位

配置示例:

```
void RCC_Configuration(void)
{
  RCC_DeInit();
  RCC_HSEConfig(RCC_HSE_ON);                          //RCC_HSE_ON——HSE晶振打开(ON)
  HSEStartUpStatus = RCC_WaitForHSEStartUp();
  if(HSEStartUpStatus == SUCCESS)                     //SUCCESS:HSE晶振稳定且就绪
  {
    RCC_HCLKConfig(RCC_SYSCLK_Div1)                   //AHB时钟为系统时钟
    RCC_PCLK2Config(RCC_HCLK_Div1);                   //APB2时钟为 HCLK
    RCC_PCLK1Config(RCC_HCLK_Div2);                   //APB1时钟为 HCLK / 2
    RCC_PLLConfig(RCC_PLLSource_HSE_Div1, RCC_PLLMul_4);
    //PLL的输入时钟 = HSE时钟频率;RCC_PLLMul_4——PLL输入时钟 x 4
    RCC_PLLCmd(ENABLE);
    while(RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET) ;
    RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK);
    //RCC_SYSCLKSource_PLLCLK——选择 PLL作为系统时钟
    while(RCC_GetSYSCLKSource() != 0x08);             //0x08:PLL作为系统时钟
  }
  //启动 GPIOA、GPIOB和 GPIOC外设时钟
  RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA | RCC_AHBPeriph_GPIOB |
                        RCC_AHBPeriph_GPIOC, ENABLE);
}
```

Chapter 5

第 5 章　通用输入输出

【导读】 通用输入输出(General Purpose Input/Output，GPIO)是嵌入式微控制器最常用、最基础、灵活性最强的控制器，可以实现简单控制，也可以组合实现复杂时序，是嵌入式程序设计必须掌握的控制器。本章首先介绍 GPIO 的引脚内部构造，不同的输入输出模式的区别，然后对 GPIO 的寄存器定义进行了详细介绍，并结合 ST 的外围控制器库函数对典型 API 进行了介绍，最后以 LED 和按键为例阐述了利用库函数进行 I/O 控制的方法。

5.1　GPIO 原理

5.1.1　GPIO 功能

GPIO 是嵌入式开发里面最基本也最常用的硬件端口，STM32L1xx 系列处理器可提供多达 128 个 I/O，实现输入输出功能，并将其分为 A～H 8 组，每组称为一个端口(PORT)，每个端口有 16 个 I/O 引脚(PIN)，每个 I/O 的速度可单独配置，最快可在 2 个时钟周期进行 I/O 翻转，支持 I/O 锁定功能、I/O 复选功能，每个引脚最多可有 16 个复选功能，最大程度的提供了灵活的 I/O 配置和使用。

I/O 的使用由 GPIO 控制器管理，每个 GPIO 端口有四个 32 位配置寄存器(GPIOx_MODER，GPIOx_OTYPER，GPIOx_OSPEEDR 和 GPIOx_PUPDR)，两个 32 位数据寄存器(GPIOx_IDR 和 GPIOx_ODR)，一个 32 位置位/复位寄存器(GPIOx_BSRR)，一个 32 位锁定寄存器(GPIOx_LCKR)和两个 32 位的复选功能选择寄存器(GPIOx_AFRH 和 GPIOx_AFRL)。

一个 I/O 端口引脚的结构如图 5-1 所示，主要由输入驱动器、输出驱动器、输入数据寄存器、输出数据寄存器和位设置/清除寄存器进行输入输出控制。

GPIO 端口的每个位可以由软件分别配置成多种模式：输入、输出、复用和模拟四种模式，每种模式下，可以通过寄存器对输入输出方式进行配置，由输入驱动器和输出驱动器的开关电路控制，每个 I/O 端口位可以自由编程，以 32 位字访问 I/O 端口寄存器。

在每个 AHB 总线时钟周期，GPIO 控制器采样 I/O 引脚的输入电平，并将输入数据存储到输入寄存器中。所有 GPIO 引脚有一个内部弱上拉和弱下拉，当配置为输入时，它们可以被激活也可以被断开；端口配置为输出模式时，GPIO 输出寄存器中的值输出到对应的

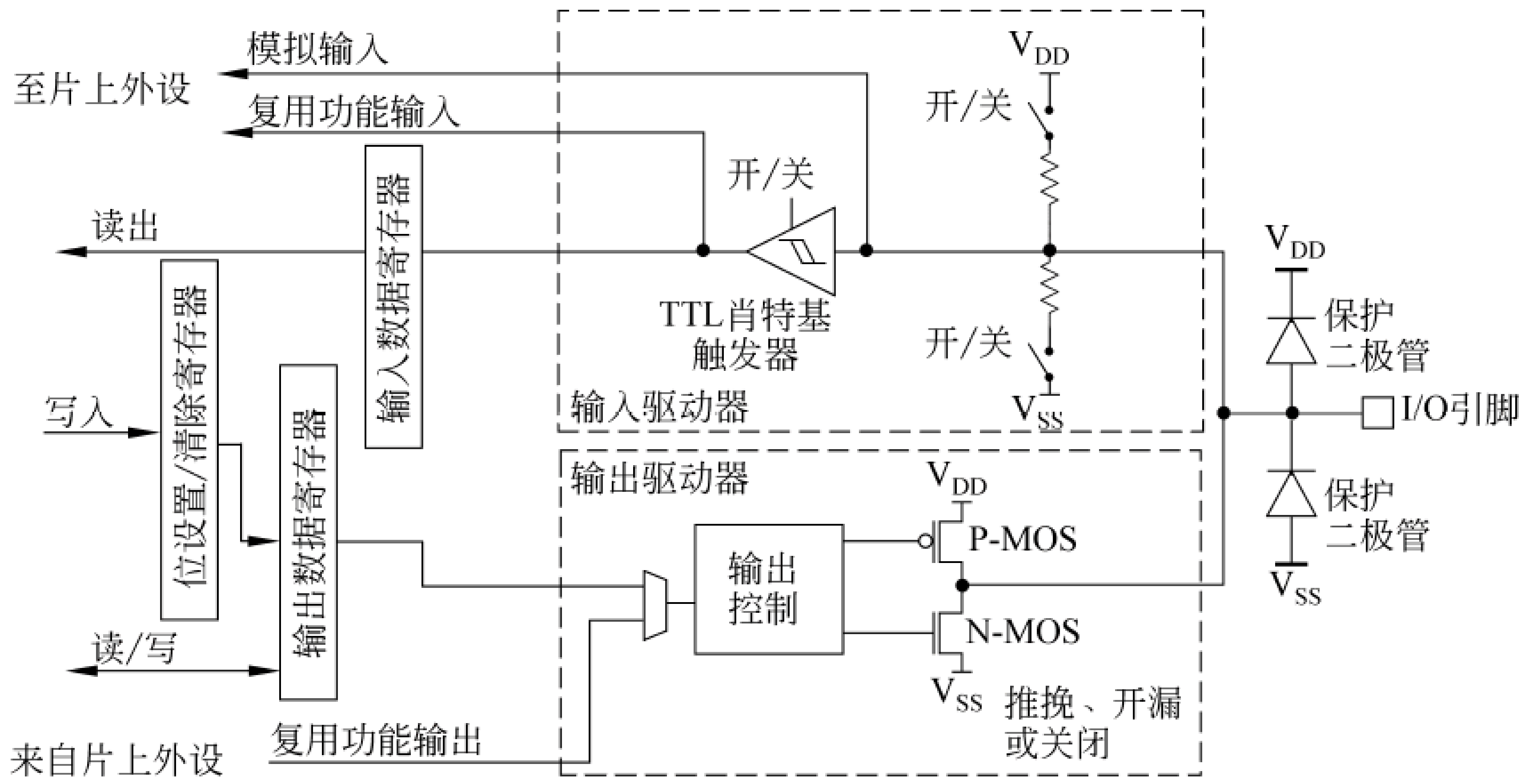


图 5-1　I/O 引脚内部电路结构

I/O 引脚上。

为便于程序对 I/O 口的输出数据寄存器控制，GPIO 控制器提供了单个或多个位的原子读写操作，通过对"置位/复位寄存器"(GPIOx_BSRR)中想要更改的位写 1 实现同时操作，无需软件进行开关中断的保护操作。

为保护 I/O 引脚配置的安全性，GPIO 控制器提供了锁定机制允许冻结 I/O 配置。当在一个端口位上执行了锁定(LOCK)程序，在下一次复位之前，将不能再更改端口位的配置。

所有的 I/O 端口及其引脚都具有复用功能，复用功能是将 GPIO 端口映射到某个外围控制器上，作为外围控制器的专用 I/O 通道，这样，可以对不使用的外围控制器的引脚当作普通 I/O 使用，当需要时配置成为专用 I/O 使用。当外围控制器功能不能满足时，我们通常使用 I/O 模拟专用控制器的时序用软件实现专用控制器的功能。每个 I/O 端口最多可以有 16 个复选功能 AF0-AF15，但只能使用一个复用功能。CPU 上电复位后，所有的 I/O 都默认使用 AF0 功能。除了特殊 I/O 引脚外(比如 JTAG 调试口(PA15、PA14、PA13、PB4、PB3)对应的 I/O 引脚 AF0 功能为 JTAG 调试端口功能，因此每个引脚被置为输入上拉或下拉模式)，I/O 引脚的 AF0 功能都是 GPIO，复位后的默认配置为浮空输入。为了使不同器件封装的外设 I/O 功能的数量达到最优，STM32L1xx 系列处理器支持复用功能重映射，可以把一些复用功能的引脚重新映射到其他一些引脚上。

所有的 GPIO 端口都有外部中断能力，将端口配置为输入模式后，可以把引脚作为中断源的输入，例如按键、事件触发等，具体在中断控制器部分进行阐述。

5.1.2　I/O 模式配置

1. 输入配置

当 I/O 端口配置为输入时：图 5-1 中的输出缓冲器被禁止，施密特触发输入被激活。

根据输入引脚配置(上拉、下拉或浮空)的不同,弱上拉或下拉电阻被连接。出现在 I/O 引脚上的数据被采样到输入数据寄存器,对输入数据寄存器的读访问可得到 I/O 引脚的状态。

浮空输入:浮空输入一般多用于外部按键输入,浮空输入状态下,I/O 的电平状态是不确定的,完全由外部输入决定。

上拉输入和下拉输入分别对应图 5-1 中的上拉电阻开关和下拉电阻开关,将输入信号的默认值保持在高电平或者低电平。

2. 输出配置

当 I/O 端口被配置为输出时,图 5-1 中的输出缓冲器被激活,施密特触发输入被激活,弱上拉或下拉电阻被禁止。输出到 I/O 引脚上的数据在每个时钟周期同时被采样到输入数据寄存器,在开漏模式时,对输入数据寄存器的读访问可得到输出 I/O 状态。在推挽式模式时,对输出数据寄存器的读访问得到当前的输出状态。

输出类型可配置为两种方式:开漏模式和推挽模式。开漏模式下,图 5-1 中的输出寄存器上的 0 激活 N-MOS,而输出寄存器上的 1 将端口置于高阻状态(P-MOS 从不被激活)。推挽模式下,图 5-1 中的输出寄存器上的 0 激活 N-MOS,而输出寄存器上的 1 将激活 P-MOS。

推挽输出:可以输出高、低电平,连接数字器件;推挽一般是指两个三极管分别受两个互补信号的控制,总是在一个三极管导通的时候另一个截止,形成推拉结构,所以导通损耗小、效率高。输出既可以向负载灌电流,也可以从负载抽取电流。推拉式输出级既提高电路的负载能力,又提高开关速度,高低电平由电源决定。

开漏输出:输出端相当于三极管的集电极,适合于做电流型的驱动,其吸收电流的能力相对强(20mA 以内)。开漏电路利用外部电路的驱动能力,可以减少芯片内部的驱动电流,一般用来连接不同电平的器件。由于开漏引脚不连接外部的上拉电阻时,只能输出低电平,如果需要同时具备输出高电平的功能,则需要接上拉电阻,上升沿的时延由上拉电阻的阻值决定,电阻小时延时就小,功耗大;反之延时大功耗小。在多个 I/O 口形成“线与”功能时,需要设置为开漏输出。

3. 复用功能配置

当 I/O 端口被配置为复用功能时,图 5-1 中,在开漏或推挽输出配置中,输出缓冲器被打开,输出缓冲器由内置外设的信号驱动输出(复用功能输出);施密特触发输入被激活,弱上拉和下拉电阻由 GPIOx_PUPDR 寄存器确定,每个时钟周期采样 I/O 引脚上的数据,读输入数据寄存器时可得到 I/O 口状态。复用功能下 I/O 口的输入输出配置以及开漏、推挽由复用功能引脚规定。

对于复用的输入功能,端口必须配置成输入模式(浮空、上拉或下拉)且输入引脚必须由外部驱动。对于复用输出功能,端口必须配置成复用功能输出模式(推挽或开漏)。对于双向复用功能,端口位必须配置复用功能输出模式(推挽或开漏)。这时,输入驱动器被配置成浮空输入模式,引脚和输出寄存器断开,并和片上外设的输出信号连接。如果软件把一个 GPIO 脚配置成复用输出功能,但是外设没有被激活,它的输出将不确定。

4. 模拟配置

当 I/O 端口被配置为模拟时，图 5-1 中的输出缓冲器被禁止，施密特触发输入被禁止，I/O 信号直接连接到处理器的模拟外围控制器电路，不经过 GPIO 控制器，施密特触发输出值被强置为 0，弱上拉和下拉电阻被禁止，此时读取输入数据寄存器时数值为 0。在低功耗休眠时，可将 I/O 设置为模拟输入状态，以降低功耗。

对于 I/O 引脚在不同应用情况下的配置如表 5-1 所示。

表 5-1　I/O 配置的典型配置

应用场景	配　置
普通 GPIO 输入	配置该引脚为浮空输入、带弱上拉输入或带弱下拉输入，不使能该引脚对应的所有复用功能模块
普通 GPIO 输出	根据需要配置该引脚为推挽输出或开漏输出，不使能该引脚对应的所有复用功能模块
普通模拟输入	配置该引脚为模拟输入模式，不使能该引脚对应的所有复用功能模块
内置外设的输入	根据需要配置该引脚为浮空输入、带弱上拉输入或带弱下拉输入，使能该引脚对应的复用功能模块
内置外设的输出	根据需要配置该引脚为复用推挽输出或复用开漏输出，使能该引脚对应的复用功能模块。

5.2　GPIO 寄存器

GPIO 控制器的寄存器如表 5-2 所示。

表 5-2　GPIO 控制器寄存器

寄存器名称	偏移量	功　能	复　位　值
端口模式寄存器(GPIOx_MODER)	0x00	配置端口的输入输出方向	端口 A：0xA800 000 端口 B：0x0000 0280 其他：0x0000 0000
端口输出类型寄存器(GPIOx_OTYPER)	0x04	配置端口输出类型	0x0000 0000
端口输出速度寄存器(GPIOx_OSPEEDR)	0x08	配置端口输出输速率	端口 B：0x0000 00C0 其他：Ox0000 0000
端口上拉下拉寄存器(GPIOx_PUPDR)	0x0C	配置端口上拉和下拉电阻	端口 A：0x6400 0000 端口 B：0x0000 0100 其他：0x0000 0000
GPIO 端口输入数据寄存器(GPIOx_IDR)	0x10	端口输入数据	0x0000 XXXX（X 表示不定态）
GPIO 端口输出寄存器(GPIOx_ODR)	0x14	端口输出数据	0x0000 0000

续表

寄存器名称	偏移量	功　　能	复　位　值
端口位设置/清除寄存器(GPIOx_BSRR)	0x18	端口输出寄存器设置为1或清为0	0x0000 0000
GPIO端口配置锁定寄存器(GPIOx_LCKR)	0x1C	锁定引脚的配置信息	0x0000 0000
复用功能低寄存器(GPIOx_AFRL)	0x20	复用功能选择0～7	0x0000 0000
复用功能高寄存器(GPIOx_AFRH)	0x28	复用功能选择8～15	0x0000 0000

1. GPIO端口模式寄存器(GPIOx_MODER)(x=A,…,H)

端口模式寄存器用于配置I/O的输入输出状态,其有效域定义如图5-2所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图5-2　端口模式寄存器定义

端口模式寄存器是一个32位寄存器,每2个bit为一组,共定义了16个I/O的端口输入输出配置。MODERy[1:0]:端口x的y引脚配置位(y=0,…,15),用于配置I/O口的输入输出方向。00表示输入模式(复位值),01表示输出模式,10表示复用功能模式,11表示模拟模式。

2. GPIO端口输出类型寄存器(GPIOx_OTYPER)(x=A,…,H)

端口输出类型寄存器用于定义I/O引脚在输出模式下是开漏输出还是推挽输出,其有效域定义如图5-3所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图5-3　端口输出类型寄存器定义

端口输出类型寄存器32位,其中高16位保留,低16位OTy表示端口x的引脚y的配置(y=0,…,15),0表示推挽输出(复位值),1表示开漏输出。

3. GPIO端口输出速度寄存器(GPIOx_OSPEEDR)(x=A,…,H)

端口输出速度寄存器是一个32位寄存器,用于配置I/O引脚的输出速度(输出信号从低电平到高电平的上升速度或高电平到低电平的下降速度),每两个bit用于确定一个I/O引脚的输出速度配置,其有效域定义如图5-4所示。

OSPEEDRy[1:0]表示端口x配置y引脚配置(y=0,…,15),00表示极低速400kHz,

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
OSPEEDR15[1:0]		OSPEEDR14[1:0]		OSPEEDR13[1:0]		OSPEEDR12[1:0]		OSPEEDR11[1:0]		OSPEEDR10[1:0]		OSPEEDR9[1:0]		OSPEEDR8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OSPEEDR7[1:0]		OSPEEDR6[1:0]		OSPEEDR5[1:0]		OSPEEDR4[1:0]		OSPEEDR3[1:0]		OSPEEDR2[1:0]		OSPEEDR1[1:0]		OSPEEDR0 1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 5-4　端口输出速度寄存器定义

01 表示低速 2MHz，10 表示中速 10MHz，11 表示高速 40MHz。

4. GPIO 端口上拉下拉寄存器(GPIOx_PUPDR)(x＝A，…，H)

端口上拉下拉寄存器用于配置 I/O 引脚是否使用内部弱上拉和弱下拉电阻，其有效域定义如图 5-5 所示。每两位 PUPDRy[1：0]表示端口 x 的 y 引脚配置(y＝0，…，15)，00 表示浮空，01 表示上拉，10 表示下拉，11 保留。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15[1:0]		PUPDR14[1:0]		PUPDR13[1:0]		PUPDR12[1:0]		PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUPDR7[1:0]		PUPDR6[1:0]		PUPDR5[1:0]		PUPDR4[1:0]		PUPDR3[1:0]		PUPDR2[1:0]		PUPDR1[1:0]		PUPDR0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 5-5　端口上拉下拉寄存器定义

5. GPIO 端口输入数据寄存器(GPIOx_IDR)(x ＝ A，…，H)

输入数据寄存器用于存储端口输入引脚的电平值，其有效域定义如图 5-6 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

图 5-6　端口输入数据寄存器定义

端口输入数据寄存器 32 位中低 16 位有效，每位分别表示对应引脚的输入电平状态。IDRy 为端口 x 的引脚 y 的输入数据 (y＝0，…，15)，该寄存器为只读寄存器，以字为单位访问。

6. GPIO 端口输出寄存器(GPIOx_ODR)(x＝A，…，H)

输出数据寄存器用于存储端口输出引脚的电平值，其有效域定义如图 5-7 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 5-7　端口输出数据寄存器定义

端口输出寄存器的高 16 位保留，低 16 位有效，每位表示对应引脚应该输出的电平状态。ODRy 表示端口 x 的引脚 y 的输出数据（y=0，…，15）。该寄存器可读写，只能以字的形式操作。为便于操作，可通过配置 GPIOx_BSRR 寄存器对指定 ODR 位设置和清除。

7. GPIO 端口位设置/清除寄存器（GPIOx_BSRR）（x = A，…，H）

端口位设置/清除寄存器用于修改数据输出寄存器的值，实现高效、原子操作的端口引脚输出电平配置，其有效域定义如图 5-8 所示。高 16 位对应 16 个 I/O 引脚置零操作，BRy 表示清除端口 x 的引脚 y（y = 0，…，15），0 表示对相应的 ODRy 位不产生影响，1 表示清除相应的 ODRy 位为 0。这些位只能以字、半字或字节写入。低 16 位对应 16 个同样的 I/O 引脚，BSy 表示设置端口 x 的引脚 y（y = 0，…，15），0 表示对相应的 ODRy 位不产生影响，1 表示设置相应的 ODRy 位为 1。读取该寄存器返回值为 0。如果同时设置了 BSy 和 BRy 的对应位，BSy 位起作用。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

图 5-8　端口输出设置/清除寄存器定义

8. GPIO 端口配置锁定寄存器（GPIOx_LCKR）（x = A，…，H）

配置锁定寄存器用于端口配置的锁定，其有效域定义如图 5-9 所示。锁定开关为寄存器的位 16 LCKK，当 LCKK 为 1 时，根据该寄存器低 16 位 LCKy 的设置对对应 I/O 引脚进行锁定。当对相应的端口位执行了 LOCK 后，在下次系统复位之前将不能再更改端口位的配置。在执行锁定序列设置期间，不可以改变 LCKP[15：0]的值，该寄存器只能以字操作。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															LCKK
															rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LCK15	LCK14	LCK13	LCK12	LCK11	LCK10	LCK9	LCK8	LCK7	LCK6	LCK5	LCK4	LCK3	LCK2	LCK1	LCK0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 5-9　端口锁定配置寄存器定义

LCKK：锁键（Lock key），该位可随时读出，但只可通过锁键写入序列修改。该位置 0 表示端口配置锁键未激活，置 1 表示端口配置锁键被激活，下次系统复位前 GPIOx_LCKR 寄存器被锁住。

LCKK 需要通过一个写入序列进行修改：写 LCKK=1 -> 写 LCKK=0 -> 写 LCKK=1 -> 读 LCKK -> 读 LCKK=1。最后一个读可省略，但可以用来确认锁键已被激活。

LCKy 表示端口 x 的 y 引脚是否加锁(y = 0,…,15),0 表示不锁定端口的配置,1 表示锁定端口的配置。这些位可读可写但只能在 LCKK 位为 0 时写入。

被锁定的寄存器包括 GPIOx_MODER,GPIOx_OTYPER,GPIOx_OSPEEDR,GPIOx_PUPDR,GPIOx_AFRL 和 GPIOx_AFRH。

9. GPIO 复用功能低寄存器(GPIOx_AFRL)(x = A,…,H)

端口复用功能寄存器用于配置 I/O 端口作为其他功能使用,其有效域定义如图 5-10 所示和 5-11 所示。每个 4 个 bit 对应一个 I/O 引脚,每个引脚最多有 16 种复用功能。端口复用功能低寄存器用于配置一个 GPIO 端口的 0~7 引脚的复用功能。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRL7[3:0]				AFRL6[3:0]				AFRL5[3:0]				AFRL4[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRL3[3:0]				AFRL2[3:0]				AFRL1[3:0]				AFRL0[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 5-10　端口复用功能低寄存器定义

AFRLy[3:0]:端口 x 的引脚 y 的复用功能启用位(y=0,…,7),AFRLy 配置的复用功能有 AFx 确定,取值如表 5-3 所示,每个引脚的 AF0~15 的具体配置见 STM32L152RE 的芯片数据手册(https://www.st.com/resource/en/datasheet/stm32l152re.pelf)

表 5-3　复用功能选择

AFRLy/AFRHy 取值	复用功能							
AFRLy	0000	0001	0010	0011	0100	0101	0110	0111
	AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7
AFRHy	1000	1001	1010	1011	1100	1101	1110	1111
	AF8	AF9	AF10	AF11	AF12	AF13	AF14	AF15

10. GPIO 复用功能高寄存器(GPIOx_AFRH)(x = A,…,H)

复用功能高寄存器用于配置 I/O 口 8~15 的复用功能,其有效域定义如图 5-11 所示,AFRHy(y=8,…,15)的配置如表 5-3 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRH15[3:0]				AFRH14[3:0]				AFRH13[3:0]				AFRH12[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRH11[3:0]				AFRH10[3:0]				AFRH9[3:0]				AFRH8[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 5-11　端口复用功能高寄存器定义

5.3 GPIO 操作函数库

CMSIS 中 GPIO 控制器的数据结构定义在 stm32lxx.h 中：

```
typedef struct
{
    __IO uint32_t MODER;       //模式寄存器,地址偏移量：0x00
    __IO uint16_t OTYPER;      //输出类型寄存器,地址偏移量：0x04
    uint16_t RESERVED0;        //保留,4字节对齐用
    __IO uint32_t OSPEEDR;     //输出速度寄存器,地址偏移量：0x08
    __IO uint32_t PUPDR;       //上拉下拉寄存器,地址偏移量：0x0C
    __IO uint16_t IDR;         //输入数据寄存器,地址偏移量：0x10
    uint16_t RESERVED1;        //保留,4字节对齐用
    __IO uint16_t ODR;         //输出数据寄存器,地址偏移量：0x14
    uint16_t RESERVED2;        //保留,4字节对齐用
    __IO uint16_t BSRRL;       //位设置/清除低寄存器,地址偏移量：0x18
    __IO uint16_t BSRRH;       //位设置/清除高寄存器,地址偏移量：0x1A
    __IO uint32_t LCKR;        //锁配置寄存器,地址偏移量：0x1C
    __IO uint32_t AFR[2];      //复用功能寄存器,地址偏移量：0x20-0x24
    __IO uint16_t BRR;         //位清除寄存器,地址偏移量：0x28,HD和XL设备有,MD没有
    uint16_t RESERVED3;        //保留,4字节对齐
} GPIO_TypeDef;
```

这样,我们可以用 GPIO_TypeDef 定义一个表示 GPIOx 的结构体变量,通过操作结构体变量对 GPIOx 的寄存器进行配置。

结构体定义中,__IO 的定义为：

```
#define  __IO   volatile
```

每个寄存器变量定义时都要使用 volatile 关键字,保证该寄存器变量存储地址的数据不会被缓存到 Cache 中。

32 位系统中,对于内存的操作以字为单位,导致结构体存在字节对齐问题,RESVEREDn 将 16 位的整型拼成 32 位整型,避免在不同编译器下字操作可能出现的问题,同时将结构体的地址排列和 GPIO 控制器的地址排列一一对应,便于通过指针对 GPIO 寄存器进行操作。

Stm32lxx.h 中对 8 个 GPIO 控制器进行了声明

```
/* !< Peripheral memory map */
#define APB1PERIPH_BASE        PERIPH_BASE
#define APB2PERIPH_BASE        (PERIPH_BASE + 0x10000)
#define AHBPERIPH_BASE         (PERIPH_BASE + 0x20000)
```

```
#define GPIOA_BASE              (AHBPERIPH_BASE + 0x0000)
#define GPIOB_BASE              (AHBPERIPH_BASE + 0x0400)
#define GPIOC_BASE              (AHBPERIPH_BASE + 0x0800)
#define GPIOD_BASE              (AHBPERIPH_BASE + 0x0C00)
#define GPIOE_BASE              (AHBPERIPH_BASE + 0x1000)
#define GPIOH_BASE              (AHBPERIPH_BASE + 0x1400)
#define GPIOF_BASE              (AHBPERIPH_BASE + 0x1800)
#define GPIOG_BASE              (AHBPERIPH_BASE + 0x1C00)
#define GPIOA                  ((GPIO_TypeDef * ) GPIOA_BASE)
#define GPIOB                  ((GPIO_TypeDef * ) GPIOB_BASE)
#define GPIOC                  ((GPIO_TypeDef * ) GPIOC_BASE)
#define GPIOD                  ((GPIO_TypeDef * ) GPIOD_BASE)
#define GPIOE                  ((GPIO_TypeDef * ) GPIOE_BASE)
#define GPIOH                  ((GPIO_TypeDef * ) GPIOH_BASE)
#define GPIOF                  ((GPIO_TypeDef * ) GPIOF_BASE)
#define GPIOG                  ((GPIO_TypeDef * ) GPIOG_BASE)
```

为便于对用户GPIO控制寄存器的配置，ST提供了GPIO标准库函数，头文件位stm32l1xx_gpio.h，程序源代码位stm32l1xx_gpio.c。在头文件中，定义了GPIO_InitTypeDef结构体用于I/O口的初始化配置，该结构体的定义如下：

```
typedef struct
{
    uint32_t GPIO_Pin;                          //需要被配置的I/O引脚
    GPIOMode_TypeDef GPIO_Mode;                 //I/O模式,用于对模式寄存器配置
    GPIOSpeed_TypeDef GPIO_Speed;               //I/O速率,用于对速率寄存器配置
    GPIOOType_TypeDef GPIO_OType;               //输出引脚类型,用于配置输出类型寄存器
    GPIOPuPd_TypeDef GPIO_PuPd;                 //上拉下拉类型,用于配置上拉下拉寄存器
}GPIO_InitTypeDef;
```

其中，I/O模式在GPIOMode_TypeDef枚举类型中定义如下：

```
typedef enum
{
    GPIO_Mode_IN   = 0x00,                      //输入模式
    GPIO_Mode_OUT  = 0x01,                      //输出模式
    GPIO_Mode_AF   = 0x02,                      //复用功能
    GPIO_Mode_AN   = 0x03                       //模拟模式
}GPIOMode_TypeDef;
```

GPIO_Pin的取值定义如下：

```
#define GPIO_Pin_0                  ((uint16_t)0x0001)
#define GPIO_Pin_1                  ((uint16_t)0x0002)
#define GPIO_Pin_2                  ((uint16_t)0x0004)
```

```
#define GPIO_Pin_3              ((uint16_t)0x0008)
#define GPIO_Pin_4              ((uint16_t)0x0010)
#define GPIO_Pin_5              ((uint16_t)0x0020)
#define GPIO_Pin_6              ((uint16_t)0x0040)
#define GPIO_Pin_7              ((uint16_t)0x0080)
#define GPIO_Pin_8              ((uint16_t)0x0100)
#define GPIO_Pin_9              ((uint16_t)0x0200)
#define GPIO_Pin_10             ((uint16_t)0x0400)
#define GPIO_Pin_11             ((uint16_t)0x0800)
#define GPIO_Pin_12             ((uint16_t)0x1000)
#define GPIO_Pin_13             ((uint16_t)0x2000)
#define GPIO_Pin_14             ((uint16_t)0x4000)
#define GPIO_Pin_15             ((uint16_t)0x8000)
#define GPIO_Pin_All            ((uint16_t)0xFFFF)
```

输出模式配置定义如下：

```
typedef enum
{   GPIO_OType_PP = 0x00,              //推挽输出
    GPIO_OType_OD = 0x01               //开漏输出
}GPIOOType_TypeDef;
```

I/O 速率的配置定义如下：

```
typedef enum
{
    GPIO_Speed_400kHz = 0x00,          //极低速
    GPIO_Speed_2MHz   = 0x01,          //低速
    GPIO_Speed_10MHz  = 0x02,          //中等速度
    GPIO_Speed_40MHz  = 0x03           //高速
}GPIOSpeed_TypeDef;
```

上拉和下拉配置定义如下：

```
typedef enum
{   GPIO_PuPd_NOPULL = 0x00,           //浮空
    GPIO_PuPd_UP     = 0x01,           //上拉
    GPIO_PuPd_DOWN   = 0x02            //下拉
}GPIOPuPd_TypeDef;
```

因此，我们对 GPIOx 的配置可以通过如下的程序实现。

```
#define GPIOA    ((GPIO_TypeDef *) GPIOA_BASE)
GPIOA->MODER = GPIO_Mode_IN;
GPIOA->OSPEEDER = GPIO_Speed_40MHz;
```

ST 提供了 GPIO 操作的函数库如表 5-4 所示。

表 5-4　GPIO 库函数列表

函　数　名	描　　述
GPIO_DeInit	将外设 GPIOx 寄存器重设为默认值
GPIO_Init	根据 GPIO_InitStruct 中指定的参数初始化外设 GPIOx 寄存器
GPIO_StructInit	把 GPIO_InitStruct 中的每一个参数按默认值填入
GPIO_ReadInputDataBit	读取指定端口引脚的输入
GPIO_ReadInputData	读取指定的 GPIO 端口输入
GPIO_ReadOutputDataBit	读取指定端口引脚的输出
GPIO_ReadOutputData	读取指定的 GPIO 端口输出
GPIO_SetBits	设置指定的数据端口位
GPIO_ResetBits	清除指定的数据端口位
GPIO_WriteBit	设置或者清除指定的数据端口位
GPIO_Write	向指定 GPIO 数据端口写入数据
GPIO_PinLockConfig	锁定 GPIO 引脚设置寄存器
GPIO_PinAFConfig	选择 GPIO 引脚的复用功能
GPIO_ToggleBits	翻转 GPIO 引脚的电平状态

1. GPIO_DeInit 函数

函数功能：将 GPIOx 的所有寄存器复位，寄存器值为默认值。

函数原型：void GPIO_DeInit (GPIO_TypeDef ＊ GPIOx)。

输入参数：GPIOx，需要复位的端口号，取值为 GPIOA、GPIOB、GPIOC、…、GPIOH。

应用示例：

```
GPIO_DeInit(GPIOA);                          //将 GPIOA端口复位
```

2. GPIO_Init 函数

函数功能：根据参数 GPIO_InitStruct 初始化 GPIOx 的寄存器。

函数原型：void GPIO_Init(GPIO_TypeDef ＊ GPIOx, GPIO_InitTypeDef ＊ GPIO_InitStruct)。

输入参数 1：GPIOx，需要配置的 I/O 端口，取值为 GPIOA、GPIOB、…、GPIOH。

输入参数 2：GPIO_InitStruct，指向已初始化成员的 GPIO_InitTypeDef 结构体变量的指针。

应用示例：

```
GPIO_InitTypeDef GPIO_InitStructure;
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_All;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
```

```
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;
GPIO_Init(GPIOA, &GPIO_InitStructure);
```

3. GPIO_PinAFConfig 函数

函数功能：设置 I/O 端口的复选功能。

函数原型：void GPIO_PinAFConfig (GPIO_TypeDef * GPIOx, uint16_t GPIO_PinSource, uint8_tGPIO_AF)。

输入参数 1：GPIOx,需要修改的 I/O 端口,取值为 GPIOA、GPIOB、…、GPIOH。

输入参数 2：GPIO_PinSource,端口的 I/O 引脚,取值为 GPIO_PinSourcex,x=0～15,其定义如下：

```
#define GPIO_PinSource0          ((uint8_t)0x00)
#define GPIO_PinSource1          ((uint8_t)0x01)
#define GPIO_PinSource2          ((uint8_t)0x02)
#define GPIO_PinSource3          ((uint8_t)0x03)
#define GPIO_PinSource4          ((uint8_t)0x04)
#define GPIO_PinSource5          ((uint8_t)0x05)
#define GPIO_PinSource6          ((uint8_t)0x06)
#define GPIO_PinSource7          ((uint8_t)0x07)
#define GPIO_PinSource8          ((uint8_t)0x08)
#define GPIO_PinSource9          ((uint8_t)0x09)
#define GPIO_PinSource10         ((uint8_t)0x0A)
#define GPIO_PinSource11         ((uint8_t)0x0B)
#define GPIO_PinSource12         ((uint8_t)0x0C)
#define GPIO_PinSource13         ((uint8_t)0x0D)
#define GPIO_PinSource14         ((uint8_t)0x0E)
#define GPIO_PinSource15         ((uint8_t)0x0F)
```

GPIO_AFSelection：复选功能选择,其取值范围为：

```
//AF 0 selection
#define GPIO_AF_RTC_50Hz       ((uint8_t)0x00)   //RTC 50/60 Hz
#define GPIO_AF_MCO            ((uint8_t)0x00)   //MCO
#define GPIO_AF_RTC_AF1        ((uint8_t)0x00)   //RTC_AF1
#define GPIO_AF_WKUP           ((uint8_t)0x00)   //Wakeup (WKUP1, WKUP2, WKUP3)
#define GPIO_AF_SWJ            ((uint8_t)0x00)   // SW、JTAG
#define GPIO_AF_TRACE          ((uint8_t)0x00)   //TRACE
//AF 1 selection
#define GPIO_AF_TIM2           ((uint8_t)0x01)   //TIM2
//AF 2 selection
#define GPIO_AF_TIM3           ((uint8_t)0x02)   //TIM3
#define GPIO_AF_TIM4           ((uint8_t)0x02)   //TIM4
#define GPIO_AF_TIM5           ((uint8_t)0x02)   // TIM5
//AF 3 selection
```

```
#define GPIO_AF_TIM9            ((uint8_t)0x03)   //TIM9
#define GPIO_AF_TIM10           ((uint8_t)0x03)   //TIM10
#define GPIO_AF_TIM11           ((uint8_t)0x03)   //TIM11
//AF 4 selection
#define GPIO_AF_I2C1            ((uint8_t)0x04)   //I2C1
#define GPIO_AF_I2C2            ((uint8_t)0x04)   // I2C2
//AF 5 selection
#define GPIO_AF_SPI1            ((uint8_t)0x05)   //SPI1
#define GPIO_AF_SPI2            ((uint8_t)0x05)   //SPI2
//AF 6 selection
#define GPIO_AF_SPI3            ((uint8_t)0x06)   //SPI3
//AF 7 selection
#define GPIO_AF_USART1          ((uint8_t)0x07)   // USART1
#define GPIO_AF_USART2          ((uint8_t)0x07)   // USART2
#define GPIO_AF_USART3          ((uint8_t)0x07)   // USART3
//AF 8 selection
#define GPIO_AF_UART4           ((uint8_t)0x08)   //UART4
#define GPIO_AF_UART5           ((uint8_t)0x08)   //UART5
//AF 10 selection
#define GPIO_AF_USB             ((uint8_t)0xA)    // USB Full speed device
//AF 11 selection
#define GPIO_AF_LCD             ((uint8_t)0x0B)   // LCD
//AF 12 selection
#define GPIO_AF_FSMC            ((uint8_t)0x0C)   // FSMC
#define GPIO_AF_SDIO            ((uint8_t)0x0C)   // SDIO
//AF 14 selection
#define GPIO_AF_RI              ((uint8_t)0x0E)   // RI
//AF 15 selection
#define GPIO_AF_EVENTOUT        ((uint8_t)0x0F)   // EVENTOUT
```

应用示例：

```
GPIO_PinAFConfig(GPIOA, GPIO_PinSource9, GPIO_AF_USART1);
```

4. GPIO_PinLockConfig 函数

函数功能：配置 I/O 锁定寄存器。

函数原型：void GPIO_PinLockConfig(GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)。

输入参数 1：GPIOx，需要配置的 I/O 端口，取值为 GPIOA、GPIOB、…、GPIOH。

输入参数 2：GPIO_Pin，需要锁定的引脚，取值为 GPIO_Pin_x，x=0～15，可以是多个引脚，GPIO_Pin 的每一位表示一个引脚号。

应用示例：

```
GPIO_PinLockConfig(GPIOA, GPIO_Pin_0 | GPIO_Pin_1);
```

5. GPIO_ReadInputData 函数

函数功能：读取指定 I/O 端口的输入数据。

函数原型：uint16_t GPIO_ReadInputData (GPIO_TypeDef * GPIOx)。

输入参数：GPIOx，所要读取的 I/O 端口，取值为 GPIOA、GPIOB、…、GPIOH。

返回值：GPIOx 的输入数据寄存器值。

应用示例：

```
uint16_t ReadValue;
ReadValue = GPIO_ReadInputData(GPIOC);
```

6. GPIO_ReadInputDataBit 函数

函数功能：读取端口指定引脚的输入数据。

函数原型：uint8_t GPIO_ReadInputDataBit (GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)。

输入参数 1：GPIOx，所要读取的 I/O 端口，取值为 GPIOA、GPIOB、…、GPIOH。

输入参数 2：GPIO_Pin，所要读取的 I/O 端口引脚号，取值为 GPIO_Pin_x，x=0～15。

返回值：该 I/O 引脚的输入数据，为 0 或 1。

应用示例：

```
uint8_t ReadValue;
ReadValue = GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7);
```

7. GPIO_ReadOutputData 函数

函数功能：读取指定 I/O 端口的输出数据寄存器值。

函数原型：uint16_t GPIO_ReadOutputData (GPIO_TypeDef * GPIOx)。

输入参数：GPIOx，所要读取的 I/O 端口，取值为 GPIOA、GPIOB、…、GPIOH。

返回值：该 I/O 端口的输出寄存器值。

应用示例：

```
uint16_t ReadValue;
ReadValue = GPIO_ReadOutputData(GPIOC);
```

8. GPIO_ReadOutputDataBit 函数

函数功能：读取指定端口指定引脚的输出数据值。

函数原型：uint8_t GPIO_ReadOutputDataBit(GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)。

输入参数 1：GPIOx，所要读取的 I/O 端口，取值为 GPIOA、GPIOB、…、GPIOH。

输入参数 2：GPIO_Pin，所要读取的指定引脚号，取值为 GPIO_Pin_x，x=0～15。

返回值：该引脚的输出数据值，为 0 或 1。

应用示例：

```
uint8_t ReadValue;
```

```
ReadValue = GPIO_ReadOutputDataBit(GPIOB, GPIO_Pin_7);
```

9. GPIO_ResetBits 函数

函数功能：清除指定 I/O 端口的指定 I/O 引脚数据。

函数原型：void GPIO_ResetBits (GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)。

输入参数 1：GPIOx,所要清除的 I/O 端口,取值为 GPIOA、GPIOB、…、GPIOH。

输入参数 2：GPIO_Pin,所要清除的 I/O 引脚,取值为 GPIO_Pin_x,x=0～15,可以是多个引脚,GPIO_Pin 的每位表示一个引脚号。

应用示例：

```
GPIO_ResetBits(GPIOA, GPIO_Pin_10 | GPIO_Pin_15);
```

10. GPIO_SetBits 函数

函数功能：设置指定 I/O 端口的指定 I/O 引脚数据。

函数原型：void GPIO_SetBits(GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)。

输入参数 1：GPIOx,所要设置的 I/O 端口,取值为 GPIOA、GPIOB、…、GPIOH。

输入参数 2：GPIO_Pin,所要设置的 I/O 引脚,取值为 GPIO_Pin_x,x=0～15,可以是多个引脚,GPIO_Pin 的每位表示一个引脚号。

应用示例：

```
GPIO_SetBits(GPIOA, GPIO_Pin_10 | GPIO_Pin_15);
```

11. GPIO_StructInit 函数

函数功能：将 GPIO_InitTypeDef 结构体变量的每个成员初始化为默认值。

函数原型：void GPIO_StructInit(GPIO_InitTypeDef * GPIO_InitStruct)。

输入参数：GPIO_InitStruct,所要初始化的 GPIO_InitTypeDef 结构体指针。

调用函数后,输入参数结构体指针的值为：

```
GPIO_Pin   = GPIO_Pin_All;
GPIO_Mode = GPIO_Mode_IN;
GPIO_Speed = GPIO_Speed_400kHz;
GPIO_OType = GPIO_OType_PP;
GPIO_PuPd = GPIO_PuPd_NOPULL;
```

应用示例：

```
GPIO_InitTypeDef GPIO_InitStructure;
GPIO_StructInit(&GPIO_InitStructure);
```

12. GPIO_ToggleBits 函数

函数功能：将指定端口的 I/O 引脚数据翻转。

函数原型：void GPIO_ToggleBits(GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)。

输入参数 1：GPIOx,所指定的 I/O 端口,取值为 GPIOA、GPIOB、…、GPIOH。

输入参数 2：GPIO_Pin,指定的 I/O 引脚,取值为 GPIO_Pin_x,x=0～15。

应用示例：

```
GPIO_ToggleBits(GPIOA, GPIO_Pin_5);
```

13. GPIO_Write 函数

函数功能：为端口数据寄存器写入指定值。

函数原型：void GPIO_Write(GPIO_TypeDef * GPIOx，uint16_t PortVal)。

输入参数 1：GPIOx，需要写入的指定端口，取值为 GPIOA、GPIOB、…、GPIOH。

输入参数 2：PortVal，需要写入到端口输出数据寄存器的值。

应用示例：

```
GPIO_Write(GPIOA, 0x1101);
```

14. GPIO_WriteBit 函数

函数功能：设置或清除 I/O 端口的一个引脚数据。

函数原型：void GPIO_WriteBit(GPIO_TypeDef * GPIOx，uint16_tGPIO_Pin，BitAction BitVal)。

输入参数 1：GPIOx，需要写入的 I/O 端口，取值为 GPIOA、GPIOB、…、GPIOH。

输入参数 2：GPIO_Pin，需要配置的制定引脚，取值为 GPIO_Pin_x，x=0～15。

输入参数 3：BitVal，指定引脚的值，取值范围为：Bit_RESET 表示引脚置 0，Bit_SET 表示引脚置 1。

应用示例：

```
GPIO_WriteBit(GPIOA, GPIO_Pin_15, Bit_SET);
```

5.4 GPIO 实例

5.4.1 GPIO 寄存器基本操作

我们可以通过 GPIO 控制器的寄存器对 I/O 进行配置。在操作 GPIO 的寄存器之前，首先要开启 GPIO 控制器的时钟，可以通过调用 RCC_AHBPeriphClockCmd (RCC_AHBPeriph_GPIOx，ENABLE)来实现，其中 RCC_AHBPeriph_GPIOx 中的 x 为要使用的 GPIO 端口号。下面以 GPIO_ReadInputData 函数、GPIO_WriteBit 函数和 GPIO_Init 函数的实现和为例阐述如何对 GPIO 的寄存器进行操作。

【例 5-1】 读输入寄存器的寄存器实现。

```
uint16_t GPIO_ReadInputData(GPIO_TypeDef * GPIOx)
{
    //直接返回 IDR 寄存器的值
    return ((uint16_t)GPIOx->IDR);
```

```
}
```

【例 5-2】 配置一个引脚的输出值的寄存器实现。

```
void GPIO_WriteBit(GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin, BitAction BitVal)
{   //Bit_RESET = 0
    if (BitVal != Bit_RESET)
    {   //直接操作 BSRRL将输出数据寄存器对应位置 1
        GPIOx->BSRRL = GPIO_Pin;
    }
    else
    {   //通过 BSRRH将输出数据寄存器对应位清 0
        GPIOx->BSRRH = GPIO_Pin ;
    }
}
```

【例 5-3】 GPIO 寄存器初始化的寄存器实现。

```
void GPIO_Init(GPIO_TypeDef * GPIOx, GPIO_InitTypeDef * GPIO_InitStruct)
{
     uint32_t pinpos = 0x00, pos = 0x00, currentpin = 0x00;
     for (pinpos = 0x00; pinpos < 16; pinpos++)
     {   //对 16个引脚进行配置
         pos = ((uint32_t)0x01) << pinpos;
         //判断是否为输入参数结构体中的引脚
         currentpin = (GPIO_InitStruct->GPIO_Pin) & pos;
         if (currentpin == pos)
         {   //如果是输入参数中的引脚,进行其他参数配置,先将 I/O引脚的模式寄存器 2bit清零
             GPIOx->MODER  &= ~ (GPIO_MODER_MODER0 << (pinpos * 2));
             //然后写入输入参数中的配置
             GPIOx->MODER |= (((uint32_t)GPIO_InitStruct->GPIO_Mode) << (pinpos * 2));
             //如果是输出模式或者复用功能,则需要设置输出速度和输出类型
             if ((GPIO_InitStruct->GPIO_Mode == GPIO_Mode_OUT) ||
                  (GPIO_InitStruct->GPIO_Mode == GPIO_Mode_AF))
             {
                //将输出速度寄存器对应的 2个 bit清 0
                GPIOx->OSPEEDR &= ~ (GPIO_OSPEEDER_OSPEEDR0 << (pinpos * 2));
                //写入新的输出速度
                GPIOx->OSPEEDR |= ((uint32_t) (GPIO_InitStruct->GPIO_Speed) << (pinpos * 2));
                //将输出类型寄存器的 1bit清 0
                GPIOx->OTYPER  &= ~ ((GPIO_OTYPER_OT_0) << ((uint16_t)pinpos)) ;
                //写入新的输出类型配置
                GPIOx->OTYPER|= (uint16_t) (((uint16_t)GPIO_InitStruct->GPIO_OType)
                                              << ((uint16_t)pinpos));
             }
```

```
            //上拉下拉寄存器的 2bit 清 0,然后将新的上拉下拉参数写入
            GPIOx->PUPDR &= ~ (GPIO_PUPDR_PUPDR0 << ((uint16_t)pinpos * 2));
            GPIOx->PUPDR |= (((uint32_t)GPIO_InitStruct->GPIO_PuPd) << (pinpos * 2));
        }
    }
}
```

程序中用到的 GPIO_PUPDR_PUPDR0、GPIO_OTYPER_OT_0、GPIO_MODER_MODER0、GPIO_OSPEEDER_OSPEEDR0 的宏定义在 stml1xx.h 中，其表示的是引脚 0 的 PUPDR、OTYPER、MODER 和 OSPEEDR 的配置域在寄存器中的位置。

```
#define GPIO_OSPEEDER_OSPEEDR0      ((uint32_t)0x00000003)
#define GPIO_OTYPER_OT_0            ((uint32_t)0x00000001)
```

5.4.2 GPIO LED 灯控制

【例 5-4】 在 STM32L152-Discovery 开发板上，通过连接到 LED 的 GPIO 引脚对绿色 LED3 灯实现反转控制。LED3 的控制引脚为 PB7，因此我们需要对 GPIOB 端口的第 7 个引脚进行配置，根据电路图，PB7 输出为 1 时灯亮，输出 0 时灯灭。

主程序代码如下：

```
#include "stm32l1xx.h"
#include "stm32L1xx_gpio.h"
#define GREEN_LED GPIO_Pin_7
#define BSRR_VAL 0x80
GPIO_InitTypeDef          GPIO_InitStructure;
void Delay(__IO uint32_t nCount)
{
    while(nCount--)
    {
    }
}
int main(void)
{
    /* GPIOD Periph clock enable */
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOB, ENABLE);
    /* Configure PB7 in output pushpull mode */
    GPIO_InitStructure.GPIO_Pin = GREEN_LED;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
```

```
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    while (1)
    {
        /* Set PB7 */
        GPIOB->BSRRL = BSRR_VAL;
        Delay(1000);
        /* Reset PB7 */
        GPIOB->BSRRH = BSRR_VAL;
        Delay(1000);
    }
}
```

程序的执行流程为：

（1）首先定义两个宏 GREEN_LED 和 BSRR_VAL，其中 GPIO_Pin_7 为系统定义的宏变量，其值为 0x80；BSRR_VAL 用于对 BSRR 寄存器进行赋值，由于需要对 GPIO_Pin_7 进行设置和清除操作，因此 BSRR_VAL 的值设为 0x80。

（2）用 GPIO_InitTypeDef 结构体类型定义一个 GPIO 寄存器结构体变量 GPIO_InitStructure，用于对 GPIOB 进行初始化。

（3）main 函数中，对结构体变量进行初始化，为 GPIOB 的第 7 个引脚设置为推挽输出，40MHz 速率，浮空，调用 GPIO_Init 函数对 GPIOB 端口的所有寄存器进行初始化。

（4）调用 RCC_AHBPeriphClockCmd 使能 GPIOB 的时钟，此时 GPIOB 可以正常工作。

（5）在 while 循环中，通过对 BRSSL 寄存器的第 7 位写 1 将 GPIOB 的输出数据寄存器第 7 位置 1，灯亮。

（6）调用 delay 函数延时。

（7）通过对 BRSSH 寄存器的第 7 位写 1 将 GPIOB 的数据寄存器第 7 位置 0，灯灭，调用 delay 函数，循环执行（5）～（7）。

根据 ST 提供的函数库，我们对 LED3 的亮灯和灭灯也可以通过 GPIO_ToggleBits(GPIOB,GREEN_LED)实现。

5.4.3　GPIO 按键输入

【例 5-5】　通过开发板按键作为输入，当按键按下时，获取按键的值，控制 LED3 变亮，按键弹起时，控制 LED3 变灭。

根据 STM32L152-Discovery 开发板的原理图，按键连接在 PA0 口上，当按键按下时，PA0 为低电平，弹起时为高电平。本例程中需要使用两个 I/O 口，PA0 作为输入，PB7 作为输出，主程序如下：

```
#include "stm32l1xx.h"
```

```
#include "stm32L1xx_gpio.h"
int main(void)
{
    GPIO_Configuration();
    while (1)
    {
        if(GPIO_ReadInputDataBit(GPIOA,GPIO_Pin_0))
            GPIO_SetBits(GPIOB, GPIO_Pin_7);
        else
            GPIO_ResetBits(GPIOB,GPIO_Pin_7);
    }
}
```

GPIO 初始化的函数实现如下：

```
void GPIO_Configuration(void)
{
    //使能 AHP 外设时钟
    RCC_AHBPeriphClockCmd( RCC_AHBPeriph_GPIOA|
                           RCC_AHBPeriph_GPIOB, ENABLE);
    //配置 LED3
    GPIO_InitTypeDef   GPIO_InitStructure;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_Init(GPIOB, &GPIO_InitStructure);
    //配置按键
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
}
```

Chapter 6

第 6 章　异常和中断处理技术

【导读】 中断是计算机管理外设和处理异常的一种重要手段。本章首先介绍中断的基本概念，对 Cortex-M3 处理器的中断向量表给出了简要说明，然后详细分析了 Cortex-M3 处理器的嵌套向量中断控制器和外部中断控制器，介绍了中断向量控制器和外部中断控制器的相应库函数，最后给出通过按键产生中断控制 LED 灯切换显示的实验代码，并对中断处理流程进行了综合分析。

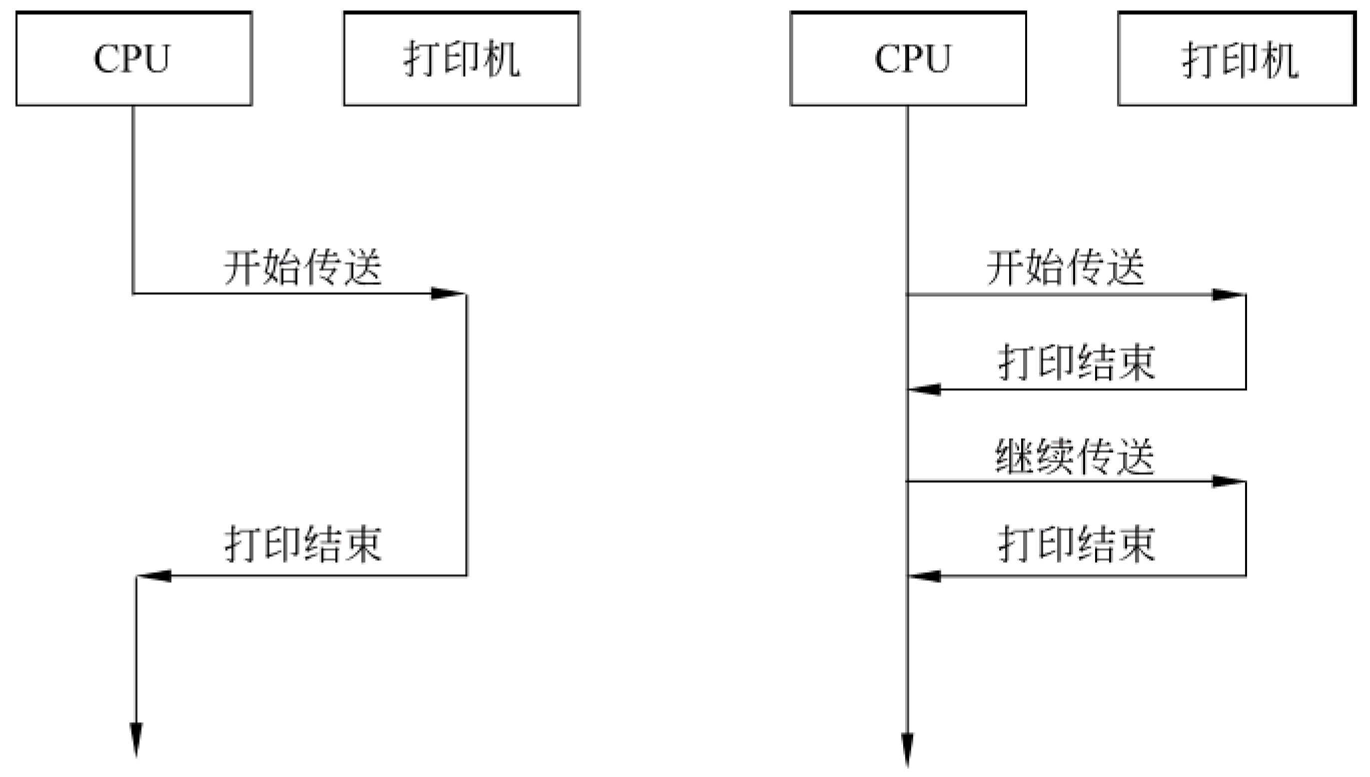
6.1　中断的基本概念

中断是计算机中的一个十分重要的概念，在现代计算机中毫无例外地都要采用中断技术。可以举一个日常生活中的例子来说明中断的概念，假如你正在厨房做饭，电话铃响了，你暂停做饭去接电话。通话完毕再继续做饭。这个例子就表现了中断及其处理过程：电话铃声使你暂时中止当前的做饭工作，而去处理更为急需处理的事情(接电话)，把急需处理的事情处理完毕之后，再回头来继续原来的事情。在这个例子中，电话铃声称为“中断请求”，暂停做饭去接电话可以称为“中断响应”，接电话的过程就是“中断处理”。相应地，计算机在执行程序的过程中，由于出现特殊情况，使得 CPU 中止正在运行的程序，而转去执行处理该特殊情况的处理程序。这个特殊情况即为中断，这个处理特殊情况的处理程序即为中断服务程序。当中断服务程序执行完毕后，再继续执行原来的程序。

计算机中采用中断技术主要是为了提高计算机系统的执行效率。图 6-1 给出了采用查询和中断两种外部设备管理方式。对于打印输出操作，CPU 工作速度和传送数据的速度都很快，而打印机打印的速度慢。如果采用查询技术，CPU 将经常处于等待状态，效率极低。而采用了中断方式后，CPU 可以进行其他的工作，只在打印机缓冲区中的当前内容打印完毕发出中断请求之后，才予以响应，暂时中断当前工作转去执行向缓冲区传送数据，传送完成后又返回执行原来的程序。

中断指的是当出现需要时，CPU 暂时停止当前程序的执行转而执行处理新情况的程序和执行过程；这种处理新情况的程序或执行过程，称为中断服务程序 ISR；中断服务程序的入口地址，称为中断向量；中断向量一般存放在计算机内存的某个特定位置，存放中断向量

的存储空间称为中断向量表;存储中断向量的地址,称为中断向量地址。



(a) 查询　　(b) 中断

图 6-1　两种外部设备工作方式

6.2　中断向量表

Cortex-M3 将打断 CPU 执行的事件分为异常与中断,其区别在于中断来自于 Cortex 内核外部,比如各种片上外设、外部中断请求等;而异常则是由于 Cortex 内核在执行指令或者访问存储等操作时所产生的。Cortex-M3 支持最多 256 个异常(广义的异常),处理器为了识别异常,对每个异常进行了唯一编号,其中编号 0～15 的为系统异常(狭义的异常),编号 16～255 的为中断,支持多达 240 个外部中断。嵌套向量中断控制器(NVIC)负责 Cortex-M3 中断管理控制,提供可屏蔽的、可嵌套的、具有动态优先级的中断管理。

【思考题:异常和中断有何区别?**】**

STM32L152 处理器的中断向量表如表 6-1 所示,每个中断或异常都有一个唯一的 IRQ 编号,异常的 IRQ 编号均为负数,中断的 IRQ 编号从 0 开始,IRQ 编号和 Cortex-M3 异常编号之间差 16。异常包括:复位 Reset、不可屏蔽中断 NMI、硬件错误 HardFault、内存管理错误 MemManage、总线错误 BusFault、指令错误 UsageFault、特权指令调用 SVCall、挂起调用 PendSV 和系统时钟 SysTick;中断包括看门狗中断,外部 I/O 中断、定时器中断等。每种异常或中断都有优先级,其中复位、NMI 和 HardFault 的优先级是固定的,其他的异常和中断的优先级均可配置,优先级值越小,优先级越高。复位中断优先级最高(－3),中断向量地址为 0x00000004,当用户按下复位键后,不论当前正在执行的是用户程序还是中断服务程序,执行控制都会转到存储在地址 0x00000004 中的程序地址去执行代码。

表 6-1　中断向量表

IRQ 编号	中断/异常编号	优先级次序	优先级类型	名称	说　明	地址
—	0	—	—	—	SP 指针初始值	0x00000000
	1	−3	固定	Reset	复位	0x00000004
−14	2	−2	固定	NMI	不可屏蔽中断	0x00000008
−13	3	−1	固定	HardFault	故障升级	0x0000000C
−12	4	0	可设置	MemManage	存储器管理故障	0x00000010
−11	5	1	可设置	BusFault	总线故障	0x00000014
−10	6	2	可设置	UsageFault	用法错误	0x00000018
—	—				保留	
−5	11	3	可设置	SVCall	SWI 系统调用	0x0000002C
−4	12	4	可设置	DebugMonitor	调试监控器	0x00000030
—	—	—			保留	0x00000034
−2	14	5	可设置	PendSV	可挂起系统服务	0x00000038
−1	15	6	可设置	SysTick	系统嘀嗒定时器	0x0000003C
0	16	7	可设置	WWDG	窗口定时器中断	0x00000040
1	17	8	可设置	PVD	电压检测中断	0x00000044
2	18	9	可设置	TAMPER	入侵检测中断	0x00000048
3	19	10	可设置	RTC	实时时钟中断	0x0000004C
4	20	11	可设置	FLASH	内存全局中断	0x00000050
5	21	12	可设置	RCC	复位和时钟控制中断	0x00000054
6	22	13	可设置	EXT0	EXT0 线中断	0x00000058
7	22	14	可设置	EXT1	EXT1 线中断	0x0000005C
8	24	15	可设置	EXT2	EXT2 线中断	0x00000060
9	25	16	可设置	EXT3	EXT3 线中断	0x00000064
10	26	17	可设置	EXT4	EXT4 线中断	0x00000068
⋮	⋮	⋮	⋮	⋮	⋮	⋮

文件 startup_stm3211xx_md.s 中，使用 DCD 指令将中断向量表填写到存储器中。当有异常或中断产生时，处理器会跳转到 DCD 后面的函数处执行中断服务程序，完整的中断向量表如下。

```
; Vector Table Mapped to Address 0 at Reset
```

```
                AREA    RESET, DATA, READONLY
                EXPORT  __Vectors
                EXPORT  __Vectors_End
                EXPORT  __Vectors_Size

__Vectors       DCD     __initial_sp                ; Top of Stack
                DCD     Reset_Handler               ; Reset Handler
                DCD     NMI_Handler                 ; NMI Handler
                DCD     HardFault_Handler           ; Hard Fault Handler
                DCD     MemManage_Handler           ; MPU Fault Handler
                DCD     BusFault_Handler            ; Bus Fault Handler
                DCD     UsageFault_Handler          ; Usage Fault Handler
                DCD     0                           ; Reserved
                DCD     0                           ; Reserved
                DCD     0                           ; Reserved
                DCD     0                           ; Reserved
                DCD     SVC_Handler                 ; SVCall Handler
                DCD     DebugMon_Handler            ; Debug Monitor Handler
                DCD     0                           ; Reserved
                DCD     PendSV_Handler              ; PendSV Handler
                DCD     SysTick_Handler             ; SysTick Handler
                ; External Interrupts
                DCD     WWDG_IRQHandler             ; Window Watchdog
                DCD     PVD_IRQHandler              ; PVD through EXTI Line detect
                DCD     TAMPER_STAMP_IRQHandler     ;Tamper and TimeStamp
                DCD     RTC_WKUP_IRQHandler         ; RTC Wakeup
                DCD     FLASH_IRQHandler            ; FLASH
                DCD     RCC_IRQHandler              ; RCC
                DCD     EXTI0_IRQHandler            ; EXTI Line 0
                DCD     EXTI1_IRQHandler            ; EXTI Line 1
                DCD     EXTI2_IRQHandler            ; EXTI Line 2
                DCD     EXTI3_IRQHandler            ; EXTI Line 3
                DCD     EXTI4_IRQHandler            ; EXTI Line 4
                DCD     DMA1_Channel1_IRQHandler    ; DMA1 Channel 1
                DCD     DMA1_Channel2_IRQHandler    ; DMA1 Channel 2
                DCD     DMA1_Channel3_IRQHandler    ; DMA1 Channel 3
                DCD     DMA1_Channel4_IRQHandler    ; DMA1 Channel 4
                DCD     DMA1_Channel5_IRQHandler    ; DMA1 Channel 5
                DCD     DMA1_Channel6_IRQHandler    ; DMA1 Channel 6
                DCD     DMA1_Channel7_IRQHandler    ; DMA1 Channel 7
                DCD     ADC1_IRQHandler             ; ADC1
```

```
            DCD     USB_HP_IRQHandler           ; USB High Priority
            DCD     USB_LP_IRQHandler           ; USB Low  Priority
            DCD     DAC_IRQHandler              ; DAC
            DCD     COMP_IRQHandler             ; COMP through EXTI Line
            DCD     EXTI9_5_IRQHandler          ; EXTI Line 9…5
            DCD     LCD_IRQHandler              ; LCD
            DCD     TIM9_IRQHandler             ; TIM9
            DCD     TIM10_IRQHandler            ; TIM10
            DCD     TIM11_IRQHandler            ; TIM11
            DCD     TIM2_IRQHandler             ; TIM2
            DCD     TIM3_IRQHandler             ; TIM3
            DCD     TIM4_IRQHandler             ; TIM4
            DCD     I2C1_EV_IRQHandler          ; I2C1 Event
            DCD     I2C1_ER_IRQHandler          ; I2C1 Error
            DCD     I2C2_EV_IRQHandler          ; I2C2 Event
            DCD     I2C2_ER_IRQHandler          ; I2C2 Error
            DCD     SPI1_IRQHandler             ; SPI1
            DCD     SPI2_IRQHandler             ; SPI2
            DCD     USART1_IRQHandler           ; USART1
            DCD     USART2_IRQHandler           ; USART2
            DCD     USART3_IRQHandler           ; USART3
            DCD     EXTI15_10_IRQHandler        ; EXTI Line 15…10
            DCD     RTC_Alarm_IRQHandler        ; RTC Alarm through EXTI Line
            DCD     USB_FS_WKUP_IRQHandler      ; USB FS Wakeup from suspend
            DCD     TIM6_IRQHandler             ; TIM6
            DCD     TIM7_IRQHandler             ; TIM7
__Vectors_End
```

文件 startup_stm3211xx_md.s 中，中断向量 Reset_Handler 对应复位中断服务程序代码如下：

```
; Reset handler routine
Reset_Handler    PROC
                 EXPORT  Reset_Handler              [WEAK]
                 IMPORT  __main
                 IMPORT  SystemInit
                 LDR     R0, =SystemInit
                 BLX     R0
                 LDR     R0, =__main
                 BX      R0
                 ENDP
```

该段代码的功能是调用 SystemInit 来配置时钟，然后进入用户的入口函数 main。

其他中断服务函数的定义在 stm32l1xxx_it.c 中：

```
//以下为系统异常服务程序,异常服务程序的实现一般为空函数或者死循环
void NMI_Handler(void)
{
    /* 不可屏蔽中断 */
}
void HardFault_Handler(void)
{
    while (1) ;      /* 硬件错误异常发生时,执行该函数,死循环 */
}
void MemManage_Handler(void)
{
    while (1) ;      /* 存储器故障异常发生时,执行该函数,死循环 */
}
void SysTick_Handler(void)
{
    /* 系统定时器异常,可以在此添加中断处理代码 */
}
…
//以下为外围控制器中断服务程序,PPP_IRQHandler 要和汇编文件中断向量表中的一致
void PPP_IRQHandler(void)
{
    /* PPP 外设的中断服务程序,可以在此添加中断处理代码 */
}
```

6.3 中断的执行过程

6.3.1 中断响应基本流程

1. 处理器模式切换

当没有异常发生时，处理器处在线程模式，当进入中断处理（ISR）或故障处理激活时，处理器将进入异常处理模式，不同类型异常处理所对应的处理器工作模式、访问级别以及栈的使用是有所不同的，也就是激活等级不同。

2. 中断响应过程

中断响应的过程为：

（1）CPU 保护现场，将返回地址、关键寄存器压栈；

（2）获取异常/中断编码；

（3）查找中断向量表，获得中断服务程序 ISR 的地址；

(4) 执行中断服务程序;

(5) 中断返回,回复现场,继续执行。

响应异常的第一个行动,就是自动保存现场的必要部分:依次把 xPSR,PC,LR,R12 以及 R0～R3 由硬件自动压入适当的堆栈中,如图 6-2 所示。如果当响应异常时,当前的代码正在使用 PSP,则压入 PSP,即使用线程堆栈;否则压入 MSP,使用主堆栈。一旦进入了中断服务程序,就将一直使用主堆栈。

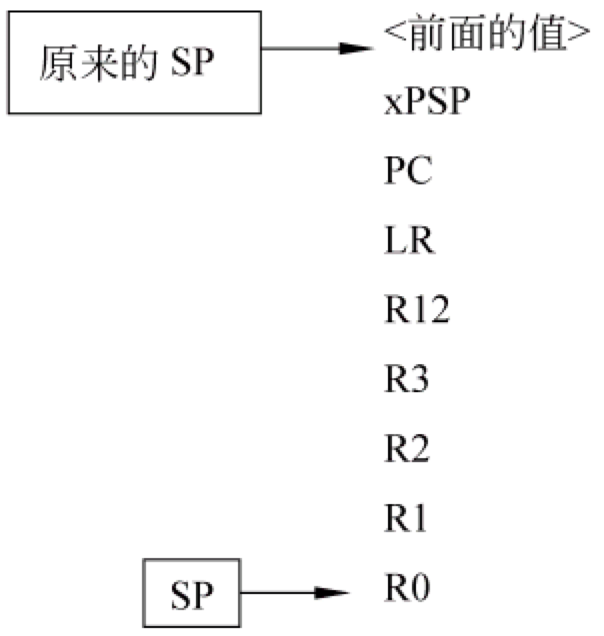


图 6-2　入栈的寄存器及其存储位置

【思考题:为何只保存 R0～R3,其他寄存器要保存吗?】

当数据总线入栈时,指令总线(ICode)同时从向量表中找出正确的异常向量,然后在服务程序的入口处预取指,入栈与取指这两个工作同时进行。

在入栈和取向量的工作都完毕之后,执行服务例程之前,还要更新一系列的寄存器。

- SP:在入栈中会把栈指针(PSP 或 MSP)更新到新的位置。在执行服务例程后,将由 MSP 负责对栈的访问。
- PSR:IPSR 会被更新为新响应的异常编号。
- PC:在向量取出完毕后,PC 将指向服务例程的入口地址。
- LR:LR 不用返回地址,而是被赋为特殊值 EXC_RETURN,在异常返回时使用。

表 6-2 为异常进入的步骤。

表 6-2　异常进入过程

动　　作	描　　述
8 个寄存器压栈	在所选的堆栈上将 xPSR,PC,R0,R1,R2,R3,R12,LR 压栈,复位异常不执行该动作
读向量表	读存储器中的向量表,地址为向量表基址+(异常号 4)。ICode 总线上的读操作能够与 DCode 总线上的寄存器压栈操作同时执行
从向量表中读 SP	只在复位时执行该动作,将 SP 更新为向量表中栈顶的值。选择堆栈,压栈和出栈之外的其他异常不能修改 SP
更新 P	利用向量表读出的位置更新 PC。直到第一条指令开始执行时,才能处理迟来异常
加载流水线	从向量表指向的位置加载指令。它与寄存器压栈操作同时执行
更新 LR	LR 设置为 EXC_RETURN,以便从异常中退出。EXC_RETURN 参见表 6-3

如表 6-4 所示,当异常服务例程执行完毕后,需要执行"异常返回"动作序列,从而恢复先前的系统状态,返回时需要使用 EXC_RETURN 的值。EXC_RETURN 只有低 4 位有效,其余位必须置为 1,其低四位的取值见表 6-3。

将先前压入栈中的寄存器出栈,栈指针恢复中断前的值。更新 NVIC 寄存器,清除中断激活位等。

表 6-3　EXC_RETURN 的返回值

EXC_RETURN[3：0]	功　　能
0b0001	返回处理模式，异常返回，获得来自主堆栈的状态，返回时指令执行使用主栈 MSP
0b1001	返回线程模式，异常返回，获得来自主堆栈的状态，返回时指令执行使用主栈 MSP
0b1101	返回线程模式，异常返回，获得来自进程堆栈的状态，返回时指令执行使用进程栈 PSP

表 6-4　异常返回过程

动　　作	描　　述
8 个寄存器出栈	如果没有被抢占，则将 PC，xPSR，R0，R1，R2，R3，R12，LR 从所选的堆栈中出栈(堆栈由 EXC_RETURN 选择)，并调整 SP
加载当前激活的中断号	加载来自被压栈的 IPSR 的位[8：0]中的当前激活的中断号。处理器用它来跟踪返回到哪个异常以及返回时清除激活位。当位[8：0]为 0 时，处理器返回线程模式
选择 SP	如果返回到异常，SP 为 SP_main，如果返回到线程模式，则 SP 为 SP_main 或 SP_process

Cortex-M3 支持由软件指定优先级，优先级范围为 0～255，0 优先级最高，255 优先级最低。为了对具有大量中断的系统加强优先级控制，NVIC 支持优先级分组机制，优先级的值分为抢占优先级区和次优先级区。我们将抢占优先级称为组优先级。如果有多个挂起异常共用相同的组优先级，则需使用次优先级区来决定同组中的异常的优先级，这就是同组内的次优先级。组优先级和次优先级的结合就是通常所说的优先级。如果两个挂起异常具有相同的优先级，则挂起异常的编号越低优先级越高。

图 6-3 为 8 位优先级 PRI_N 的占先优先级区(x)和次优先级区(y)的配置。

PRIGROUP[2:0]	中断优先级区，PRI_N[7:0]				
	二进制点的位置	占先区	次优先级区	占先优先级的数目	次优先级的数目
b000	bxxxxxxx.y	[7:1]	[0]	128	2
b001	bxxxxxx.yy	[7:2]	[1:0]	64	4
b010	bxxxxx.yyy	[7:3]	[2:0]	32	8
b011	bxxxx.yyyy	[7:4]	[3:0]	16	16
b100	bxxx.yyyyy	[7:5]	[4:0]	8	32
b101	bxx.yyyyyy	[7:6]	[5:0]	4	64
b110	bx.yyyyyyy	[7]	[6:0]	2	128
b111	b.yyyyyyyy	无	[7:0]	0	256

图 6-3　中断优先级配置

高抢占优先级的中断会打断当前用户程序或当前正在执行的低抢占优先级的中断服务程序，即中断嵌套；在抢占优先级相同的情况下，次优先级高的中断优先被响应。但是，在抢

占优先级相同的情况下，如果有低次优先级的中断服务程序正在执行，则高次优先级的中断不能打断，即不能嵌套中断。当两中个断源的抢占式优先级相同时，这两个中断没有嵌套关系，当一个中断到来后如果另一个中断正在处理，这个后到的中断就要等到前一个中断处理完之后才能被处理。如果这两个中断同时到达，中断控制器则根据它们的响应优先级高低来决定先处理哪一个；如果它们的抢占式优先和次优先级都相等，则根据它们在中断表中的排位顺序决定先处理哪一个。

6.3.2　中断优化技术

为提高中断的响应速度，在优先级的基础上，Cortex-M3 提供了以下中断优化技术。

1. 抢占

当新的异常比当前异常有更高优先级时，则中断当前操作流程，响应新的异常并执行其 ISR，即发生中断嵌套，如图 6-4 所示。

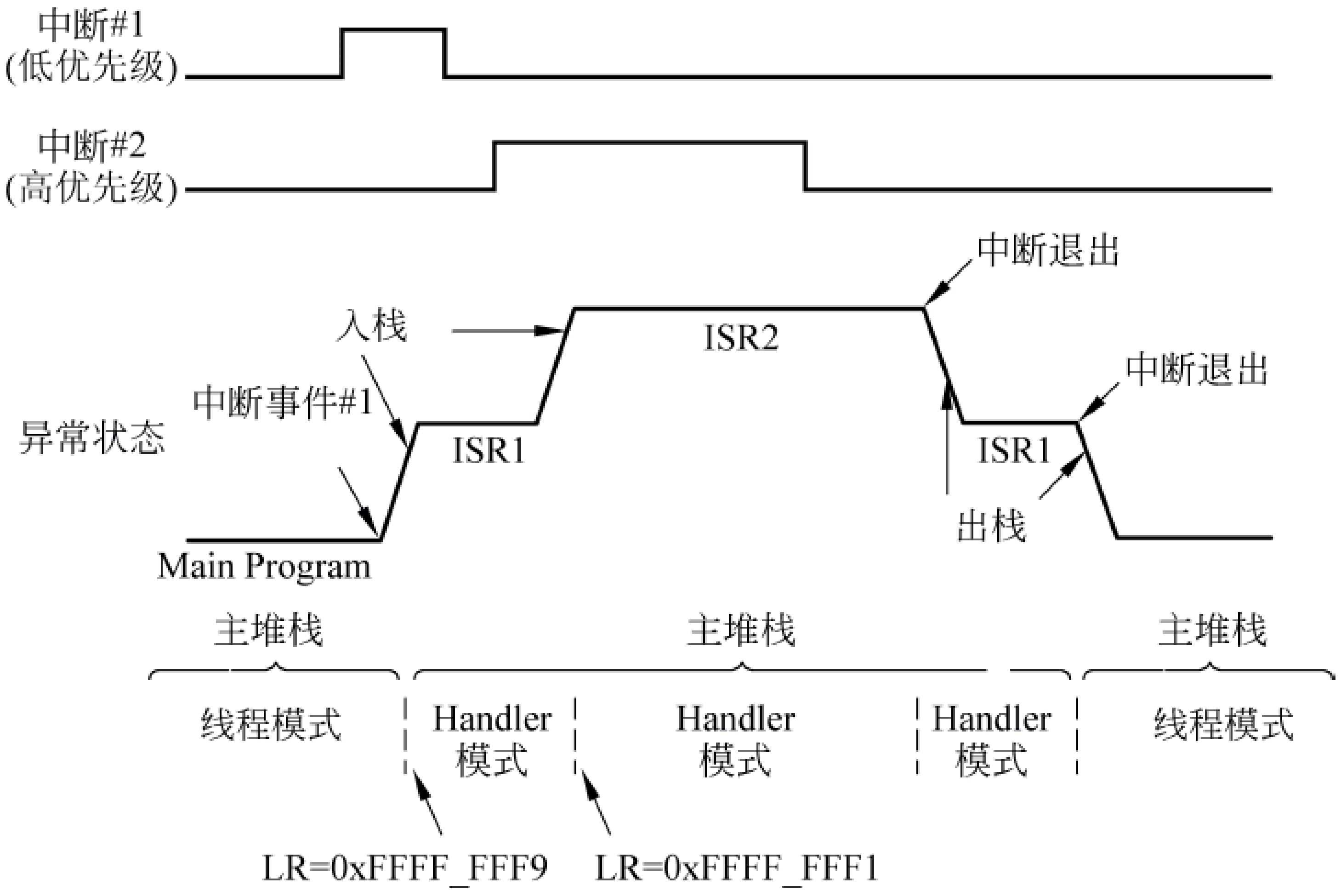


图 6-4　抢占过程

2. 末尾连锁

末尾连锁能够在两个中断之间没有多余的状态保存和恢复指令的情况下实现背对背处理。在退出 ISR 并进入另一个中断时，处理器略过 8 个寄存器的出栈和压栈操作，因为它对堆栈的内容没有影响。如果挂起中断的优先级比所有被压栈的异常的优先级都高，则处理器执行末尾连锁直接取出挂起中断的向量，在退出前一个 ISR 之后六个周期，开始执行被末尾连锁的 ISR，如图 6-5 所示。

3. 返回

如果挂起的异常中没有比栈中的 ISR 异常优先级更高的，则处理器执行返回操作，恢复进入 ISR 之前的状态，在恢复现场的过程中，如果此时有更高优先级的中断到来，但处理

器还没有恢复完成现场，此时放弃恢复现场的过程，直接将更高优先级的中断作为尾链处理。

【思考题：已经弹出栈的值如何处理?**】**

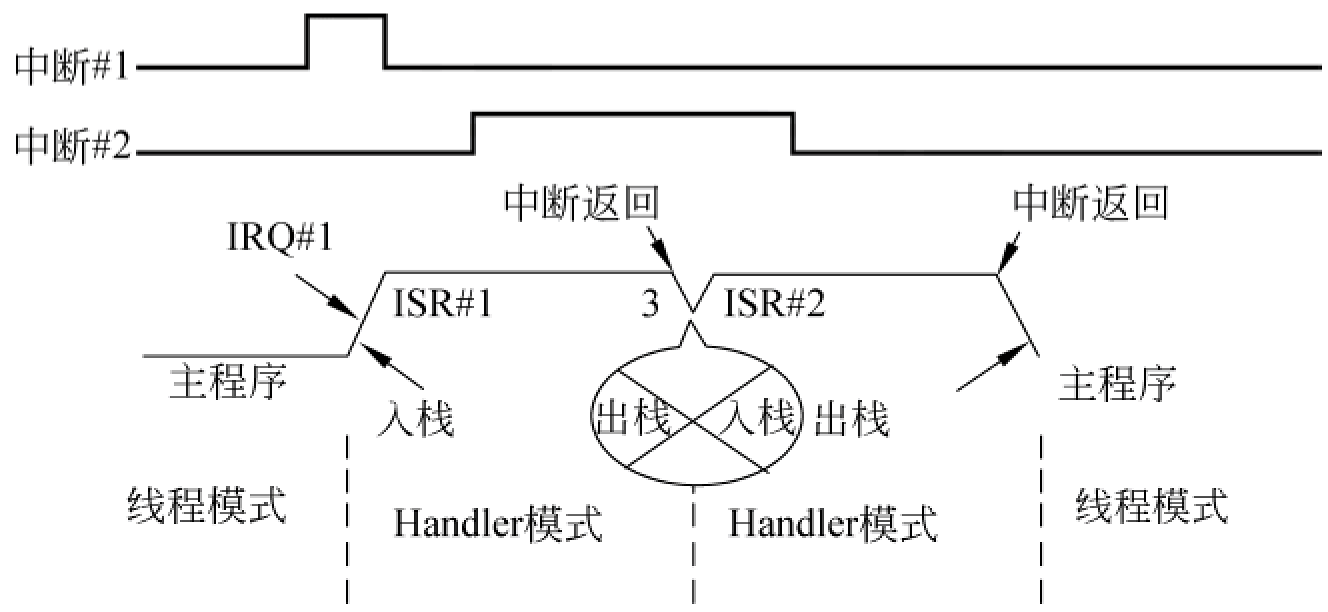


图 6-5　尾链处理过程

4. 迟到

如果前一个 ISR 还没有进入执行阶段，并且迟来中断的优先级比前一个中断的优先级要高，则迟来中断能够抢占前一个中断，如图 6-6 所示。响应迟来中断时需执行新的取向量地址和 ISR 预取操作。迟来中断不保存状态，因为状态保存已经被最初的中断执行过了，因此不需要重复执行。

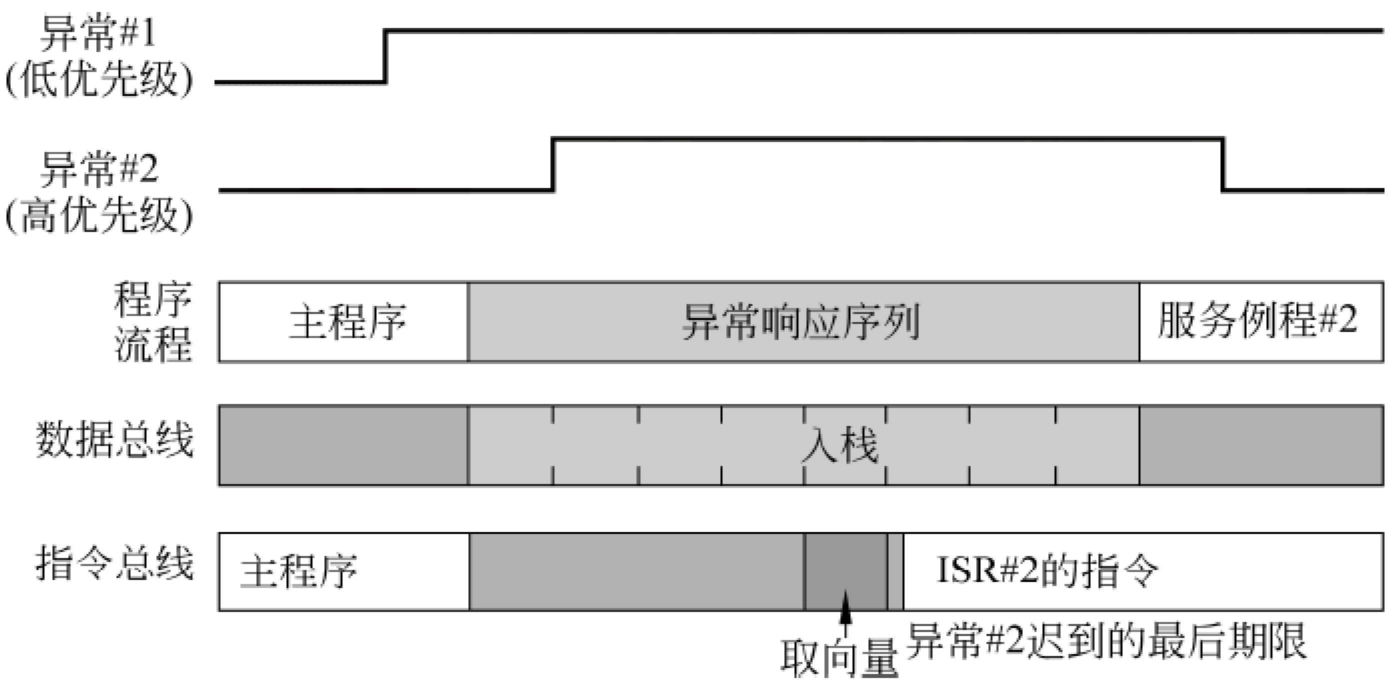


图 6-6　迟到中断响应过程

6.3.3　系统异常

1. 复位

表 6-5 为复位的过程。复位后，CM3 读取下列两个 32 位整数的值。

(1) 从地址 0x0000,0000 处取出 MSP 的初始值。

(2) 从地址 0x0000,0004 处取出 PC 的初始值(复位异常处理函数)，然后从这个值所对应的地址处取指，如图 6-7 所示。

表 6-5　复位动作

动　　作	描　　述
NVIC 复位，内核保持在复位状态	NVIC 对它的大部分寄存器进行清零。处理器位于线程模式，优先级为特权模式，堆栈设置为主堆栈
NVIC 将内核从复位状态释放	NVIC 将内核从复位状态释放
内核设置堆栈	内核从向量表偏移 0 中读取最初的 MSP 值
内核设置 PC 和 LR	内核从向量表偏移中读取最初的 PC。LR 设置为 0xFFFFFFFF
运行复位程序	NVIC 的中断被禁止，NMI 和硬件故障未被禁止

图 6-7　复位异常处理过程

2. 故障

故障时处理器内部产生的异常，能够产生中止故障的 4 个事件包括：

(1) 指令取指或向量表加载时的总线错误；

(2) 数据访问时的总线错误；

(3) 内部检测到的错误，例如未定义的指令或试图用 BX 指令来改变状态；

(4) 由于违反了特权模式或未管理的区域而引起的 MPU 故障。

① 存储器故障：触犯了 MPU 设置的保护规范和某些非法访问引起的存储器错误，例如访问了所有 MPU 覆盖范围之外的地址，访问了没有存储器与之对应的空地址，往只读区写数据，用户级下访问了只允许在特权级下访问的地址等。

② 总线故障：当 AHB 接口上正在传送数据时，如果回复了一个错误信号，则会产生总线故障，例如取指过程中的预取流产，数据读写过程中的数据流产等。

③ 用法故障：错误使用导致的故障，如执行了协处理器指令(Cortex-M3 没有协处理器)、执行了未定义的指令、尝试进入 ARM 状态、无效的中断返回(LR 中包含了无效/错误的值)、使用多重加载/存储指令时，地址没有对齐等。

④ 硬件故障：上述常规故障处理中发生问题时，故障升级成为硬件故障。

3. SVC 系统调用和 PendSV 挂起调用

SVC(系统服务调用)和 PendSV(挂起系统调用)异常多用于在操作系统。SVC 用于产生系统函数的调用请求，从用户级切入到特权级，如用户程序使用 SVC 发出对系统服务函数的呼叫请求，以这种方法调用它们来间接访问硬件。PendSV 可以像普通的中断一样被悬起的，可以让其他重要的任务完成后才执行该异常处理，PendSV 的典型使用场合是操作系统的上下文切换时(在不同任务之间切换)。

4. SysTick 定时器

SysTick 定时器被捆绑在 NVIC 中，用于产生 SYSTICK 异常。大多操作系统需要一个

硬件定时器来产生操作系统需要的滴答中断，作为整个系统的时基，以维持操作系统进程调度，Cortex-M3 处理器内部包含了一个简单的定时器。所有的 Cortex-M3 芯片都带有 SysTiuck 定时器这样使得软件在不同 Cortex-M3 器件间的移植工作得以化简。

6.4 嵌套向量中断控制器 NVIC

6.4.1 STM32L152 NVIC

Cortex-M3 处理器中定义了 8 个位来设置中断源的优先级，STM32L152 实际只使用了其中的高 4 位，低 4 位置为 0，如表 6-6 所示。分别有 0～4 共 5 组中断优先级方式。例如，对于分组 3 的方式，抢占优先级占据 3 位，则一共有 0～7 共 8 个抢占优先级，而从优先级只占 1 位，则一共只有 0 和 1 共两个次优先级。

表 6-6 STM32L152 中断优先级分组

分　组	抢占优先级位数和取值范围	从优先级位数和取值范围
0	0(无)	4(0～15)
1	1(0～1)	3(0～7)
2	2(0～3)	2(0～3)
3	3(0～7)	1(0～1)
4	4(0～15)	0(无)

STM32L152 在实现 NVIC 时，根据具体情况对 Cortex-M3 的 NVIC 进行了配置，其特点是：最多支持 81 个外部中断(Cortex-M3 支持 240 个)，16 种优先级。

【思考题：如果 STM32L152 配置分组为 1，抢占优先级为 1，从优先级为 4，那么优先级的值为多少?**】**

6.4.2 NVIC 寄存器

NVIC 控制器包含在 Cortex-M3 内核中，除了中断和系统异常控制外，还用来实现系统控制。其寄存器列表如表 6-7 所示。

表 6-7 NVIC 寄存器

名　称	类型	地址	复位值
中断控制类型寄存器	只读	0xE000E004	由配置定义
IRQ0～IRQ31 使能设置寄存器	读写	0xE000E100	0x00000000

续表

名　　称	类型	地址	复位值
⋮	⋮	⋮	⋮
IRQ224～IRQ239 使能设置寄存器	读写	0xE000E11C	0x00000000
IRQ0～IRQ31 使能清除寄存器	读写	0xE000E180	0x00000000
⋮	⋮	⋮	⋮
IRQ224～IRQ239 使能清除寄存器	读写	0xE000E19C	0x00000000
IRQ0～IRQ31 挂起设置寄存器	读写	0xE000E200	0x00000000
⋮	⋮	⋮	⋮
IRQ224～IRQ239 挂起设置寄存器	读写	0xE000E21C	0x00000000
IRQ0～IRQ31 挂起清除寄存器	读写	0xE000E280	0x00000000
⋮	⋮	⋮	⋮
IRQ224～IRQ239 挂起清除寄存器	读写	0xE000E29C	0x00000000
IRQ0～IRQ31 激活位寄存器	只读	0xE000E29C	0x00000000
⋮	⋮	⋮	⋮
IRQ224～IRQ239 激活位寄存器	只读	0xE000E31C	0x00000000
IRQ0～IRQ31 优先级寄存器	读写	0xE000E400	0x00000000
⋮	⋮	⋮	⋮
IRQ236～IRQ239 优先级寄存器	读写	0xE000E4F0	0x00000000
中断控制状态寄存器	读写或只读	0xE000ED04	0x00000000
向量表偏移寄存器	读写	0xE000ED08	0x00000000
应用中断/复位控制寄存器	读写	0xE000ED0C	0x00000000
系统控制寄存器	读写	0xE000ED10	0x00000000
配置控制寄存器	读写	0xE000ED14	0x00000000
系统处理器 4～7 优先级寄存器	读写	0xE000ED18	0x00000000
系统处理器 8～11 优先级寄存器	读写	0xE000ED1C	0x00000000
系统处理器 12～15 优先级寄存器	读写	0xE000ED20	0x00000000
系统处理器控制与状态寄存器	读写	0xE000ED24	0x00000000
可配置故障状态寄存器	读写	0xE000ED28	0x00000000
硬故障状态寄存器	读写	0xE000ED2C	0x00000000
调试故障状态寄存器	读写	0xE000ED30	0x00000000
存储器管理地址寄存器	读写	0xE000ED34	不可预测

续表

名　　称	类型	地址	复位值
总线故障地址寄存器	读写	0xE000ED38	不可预测
PFR0 处理器功能寄存器 0	只读	0xE000ED40	0x00000000
PFR1 处理器功能寄存器 1	只读	0xE000ED44	0x00000000
DFR0 调试功能寄存器 0	只读	0xE000ED48	0x00000000
AFR0 辅助功能寄存器 0	只读	0xE000ED4C	0x00000000
MMFR0 存储器模型功能寄存器 0	只读	0xE000ED50	0x00000000
MMFR1 存储器模型功能寄存器 1	只读	0xE000ED54	0x00000000
MMFR2 存储器模型功能寄存器 2	只读	0xE000ED58	0x00000000
MMFR3 存储器模型功能寄存器 3	只读	0xE000ED5C	0x00000000
ISAR0 ISA 功能寄存器 0	只读	0xE000ED60	0x01141110
ISAR1 ISA 功能寄存器 1	只读	0xE000ED64	0x02111000
ISAR2 ISA 功能寄存器 2	只读	0xE000ED68	0x21112231
ISAR3 ISA 功能寄存器 3	只读	0xE000ED6C	0x01111110
ISAR4 ISA 功能寄存器 4	只读	0xE000ED70	0x01310102
软件触发中断寄存器	只写	0xE000EF00	—

在 NVIC 异常中断控制中，我们主要关注 Cortex-M3 外部的 240 个中断，这些中断的配置主要包括 5 类寄存器：中断使能寄存器 ISER、中断清除寄存器 ICER、中断挂起寄存器 ISPR、中断挂起清除寄存器 ICPR、中断激活寄存器 IABR 和中断优先级寄存器 IPR。其中 ISER 和 ICER 用于中断屏蔽管理，即是否允许中断请求；ISPR 和 ICPR 用于中断挂起管理，设置挂起或清除挂起状态；这两组寄存器各有 8 个，STM32L152 只支持 81 个外部中断，因此中断屏蔽和中断挂起只有 3 对寄存器，分别为 ISER0～ISER2、ICER0～ICER2、ISPR0～ISPR2、ICPR0～ICPR2。IABR 有 3 个寄存器 IABR0～IABR2，IPR 有 81 个 8bit 的寄存器 IPR0～IPR80。所有的 NVIC 寄存器都可采用字节、半字和字方式进行访问，不管处理器存储字节的顺序如何，所有 NVIC 寄存器和系统调试寄存器都是采用小端(little endian)字节排列顺序，即低位字节存储在低地址。另外，下列寄存器也对中断处理有重大影响：

- 全局异常屏蔽寄存器(PRIMASK，FAULTMASK 以及 BASEPRI)；
- 向量表偏移量寄存器；
- 优先级分组位段；
- 软件触发中断寄存器。

1）中断使能寄存器（NVIC_ISERx)，x＝0，1，2

NVIC 支持可屏蔽中断，即可以通过控制中断使能寄存器和中断清除寄存器对每个外部中断源进行独立的控制。中断使能寄存器的有效域定义如图 6-8 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SETENA[31:16]															
rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SETENA[15:0]															
rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs

图 6-8　中断使能寄存器

中断使能位 SETENA[31：0]，写 1 使能中断，写 0 无效。读该寄存器时，对应位为 1 表示该中断已使能，0 表示该中断没有使能。

2）中断清除寄存器（NVIC_ICERx），x＝0，1，2

中断清除寄存器的有效域定义如图 6-9 所示。中断清除位 CLRENA[31：0]，写 1 表示禁止该中断，写 0 无效；读取该寄存器，对应位为 1 表明相应的中断被使能，否则被禁用。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CLRENA[31:16]															
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CLRENA[15:0]															
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

图 6-9　中断清除寄存器

3）中断挂起设置寄存器（NVIC_ISPRx），x＝0，1，2

一个发生的中断如果不能被立即响应，就称它被挂起，被挂起的中断由挂起状态寄存器保持，保证中断源释放了中断请求信号后中断请求也不会丢失。

中断挂起寄存器的有效域定义如图 6-10 所示。中断挂起设置位 SETPEND[31：0]，写 1 则相应中断被挂起，写 0 无效，读取该寄存器的值，对应位为 1 表明该中断正在挂起中。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SETPEND[31:16]															
rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SETPEND[15:0]															
rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs

图 6-10　中断挂起设置寄存器

当一个中断正在挂起时，写 NVIC_ISPRx 寄存器的相应位为 1 对该中断没有影响；当写入 NVIC_ISPRx 寄存器相应位为 1 但该中断被禁用时，将相应中断置为挂起状态。

4）中断挂起清除寄存器（NVIC_ICPRx），x＝0，1，2

中断挂起清除寄存器的有效域定义如图 6-11 所示。中断挂起清除位 CLRPEND[31：0]，写 1 时清楚挂起的中断，写 0 无效；读该寄存器，相应位为 1 时表明中断正在挂起中。

5）中断活动寄存器(NVIC_IABRx)，x＝0，1，2

中断活动寄存器的有效域定义如图 6-12 所示。中断激活状态 ACTIVE[31：0]，1 表示中断处于激活状态或激活挂起状态，0 表示中断没有被激活。如果一个挂起的中断被使能，

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CLRPEND[31:16]															
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CLRPEND[15:0]															
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

图 6-11　中断挂起设置寄存器

NVIC 将激活该中断。每个外部中断都有一个活动状态位。在处理器执行了其 ISR 的第一条指令后，它的活动位就被置 1，并且直到 ISR 返回时才硬件清零。由于支持嵌套，允许高优先级异常抢占某个 ISR。然而，哪怕一个中断被抢占，其活动状态也依然为 1。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ACTIVE[31:16]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ACTIVE[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

图 6-12　中断活动寄存器

6）中断优先级寄存器（NVIC_IPRx），x＝0，1，…，80

每一个中断优先级寄存器 IPRx 包括 1 个中断的优先级，优先级的设置参见表 6-2 中断优先级分组，每个 IPRx 寄存器是一个字节，如图 6-13 所示。STM32 只使用 8bit 中的 4 个比特，即 IPRx 的低 4 位为 0。

	31　　24	23　　16	15　　8	7　　0
E000E400	PRI_3	PRI_2	PRI_1	PRI_0
E000E404	PRI_7	PRI_6	PRI_5	PRI_4
E000E408	PRI_11	PRI_10	PRI_9	PRI_8
E000E40C	PRI_15	PRI_14	PRI_13	PRI_12
E000E410	PRI_19	PRI_18	PRI_17	PRI_16
E000E414	PRI_23	PRI_22	PRI_21	PRI_20
E000E418	PRI_27	PRI_26	PRI_25	PRI_24
E000E41C	PRI_31	PRI_30	PRI_29	PRI_28

图 6-13　中断优先级寄存器

7）软件中断触发寄存器（NVIC_STIR）

软件触发中断寄存器的有效域定义如图 6-14 所示。中断号 NTID[8：0]，用于指示软

件产生的中断编号。INTID[8：0]中写入 0～239 的数值，产生一个对应于该编号的中断。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Reserved							INTID[8:0]								
							w	w	w	w	w	w	w	w	w

图 6-14　软件触发中断寄存器

【思考题：为何中断使能、清除，中断挂起、挂起清除要使用成对的寄存器控制?】

当有中断发生时，外围设备将发送中断信号给 NVIC，如果 NVIC 收到中断信号且该中断被使能，但中断没有被激活，将该中断置到挂起状态；当处理器响应该中断并执行中断服务程序时，挂起状态的中断被激活，ISR 执行完成后，中断状态被置为没有激活状态；若在 ISR 执行过程中，中断信号再次到达，ISR 执行完成后，将该中断置为挂起状态，等待再次响应该中断。

6.4.3　系统异常处理

系统异常指的是异常向量表前 15 个异常，NVIC 中，系统异常的处理不采用上述的 5 类寄存器，而是由中断控制状态寄存器 ICSR、应用中断/复位控制寄存器 AIRCR、系统控制寄存器 SCR、配置控制寄存器 CCR、系统异常 4～7 优先级寄存器 SHPR1、系统异常 8～11 优先级寄存器 SHPR2、系统异常 12～15 优先级寄存器 SHPR3、系统异常控制与状态寄存器 SHCSR 等对中断使能、清除、挂起设置、挂起清除、优先级、激活状态等进行控制和描述。

其中，应用中断与复位控制寄存器 AIRCR 用于决定数据的字节顺序、清除所有有效的状态信息、执行系统复位、改变优先级分组位置等功能，其寄存器定义如图 6-15 所示。

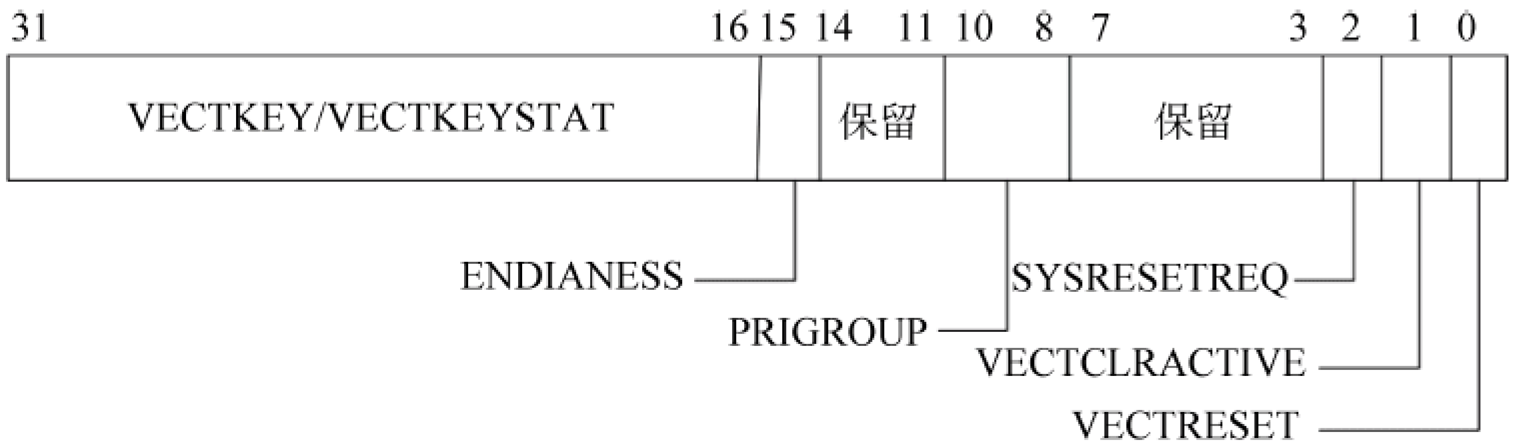


图 6-15　应用中断与复位控制寄存器

PRIGROUP 用于配置中断优先级分组，其取值为：

0　7.1 表示 7 位抢占式优先级，1 位子优先级；

1　6.2 表示 6 位抢占式优先级，2 位子优先级；

2　5.3 表示 5 位抢占式优先级，3 位子优先级；

3　4.4 表示 4 位抢占式优先级，4 位子优先级；

4　3.5 表示 3 位抢占式优先级，5 位子优先级；

5　2.6 表示 2 位抢占式优先级，6 位子优先级；

6　1.7 表示 1 位抢占式优先级，7 位子优先级；

7　0.8 表示 0 位抢占式优先级，8 位子优先级。

用法 Fault，总线 Fault 以及存储器管理 Fault 都是特殊的异常，它们的使能、挂起和激活控制是通过系统异常控制及状态寄存器(SHCSR)实现的。此外，SHCSR 寄存器还用于管理 SVC、SysTick、SendSV、监控异常的激活状态。NMI、SysTick 定时器以及 PendSV 的挂起通过 ICSR 寄存器配置，NMI 和硬件错误无需使能就可以发生。

系统异常的优先级由 SHPR1～SHPR3 寄存器设置，可设置的优先级为 0～31。SHPR1～SHPR3 可按字节访问，如表 6-8 所示。

表 6-8　系统异常优先级寄存器

地　　址	名　　称	说　　明
0xE000_ED18	PRI_4	存储器管理 Fault 的优先级
0xE000_ED19	PRI_5	总线 Fault 的优先级
0xE000_ED1A	PRI_6	用法 Fault 的优先级
0xE000_ED1F	PRI_11	SVC 优先级
0xE000_ED20	PRI_12	调试监控优先级
0xE000_ED22	PRI_14	PendSV 的优先级
0xE000_ED23	PRI_15	SysTick 的优先级

6.4.4　全局中断管理

Cortex-M3 的异常/中断屏蔽寄存器组有 3 个寄存器用于全局中断管理。

- PRIMASK 寄存器只有最低位有效，该寄存器置为 1 后，就关掉所有可屏蔽异常，只剩下 NMI 和硬件 Fault 可以响应。其默认值是 0，表示没有关闭中断。
- FAULTMASK 寄存器为单一比特的寄存器。置为 1 后，只有 NMI 可以响应。默认值为 0，表示没有关异常。
- BASEPRI 寄存器低 9 位有效，定义了被屏蔽优先级的阈值。当它被设置为某个值后，所有优先级编号大于或等于此值的中断(即优先级化中断)都被关。若设置成 0，则不关断任何中断，0 为默认值。

我们可以使用 MRS/MSR 指令访问这三个寄存器：

```
MRS     R0, BASEPRI     ;读取 BASEPRI 到 R0 中
MOV     R1, #20
MSR     BASEPRI,R1      ;将 20 数据写入到 BASEPRI 中
```

Cortex-M3 还专门设置了 CPS 指令用于 PRIMASK 和 FAULTMASK 的操作：

```
CPSID    I     ;PRIMASK=1,关中断
```

```
CPSIE    I    ;PRIMASK=0,开中断
CPSID    F    ;FAULTMASK=1,关异常
CPSIE    F    ;FAULTMASK=0,开异常
```

在 CMSIS 库中，core_cmFunc.h 中提供了以下函数用于上述寄存器的操作：

- void __set_BASEPRI(uint32_t basePri)；
- uint32_t __get_BASEPRI(void)；
- void __set_PRIMASK(uint32_t priMask)；
- uint32_t __get_PRIMASK(void)；
- void __set_FAULTMASK(uint32_t faultMask)；
- uint32_t __get_FAULTMASK(void)；

针对 C 语言，我们常用的两个全局中断开关函数为：

- void __disable_irq(void)；
- void __enable_irq(void)；

中断向量表默认存储在 0 地址，0 地址是 ROM 区，CM3 允许向量表重定位，中断向量表可以放置到 RAM 区，这样可以方便在程序中对中断向量表进行修改。NVIC 控制器的向量表偏移量寄存器 VTOR(地址 0xE000ED08)用于配置中断向量表的存储地址。VTOR 的定义如表 6-9 所示。

表 6-9 偏移量寄存器的定义

位段	名称	类型	复位值	描　述
29	TBLBASE	RW	0	向量表在 Code 区(0)，还是在 RAM 区(1)
15	ENDIANESS	R	—	向量表的起始地址，一般置为 0

6.4.5 NVIC 库函数

NVIC 库函数提供使能或者失能 IRQ 中断，使能或者失能单独的 IRQ 通道，改变 IRQ 通道的优先级等功能。

CMSIS 库中，NVIC 寄存器结构 NVIC_TypeDeff 在文件 core_cm3.h 中的定义如下：

```
typedef struct
{
    __IO uint32_t ISER[8];
    uint32_t RESERVED0[24];
    __IO uint32_t ICER[8];
    uint32_t RSERVED1[24];
    __IO uint32_t ISPR[8];
    uint32_t RESERVED2[24];
    __IO uint32_t ICPR[8];
```

```
    uint32_t RESERVED3[24];
    __IO uint32_t IABR[8];
    uint32_t RESERVED4[56];
    __IO uint8_t  IP[240];
    uint32_t RESERVED5[644];
    __O  uint32_t STIR;
} NVIC_Type;
```

操作 NVIC 的指针定义如下：

```
#define SCS_BASE    (0xE000E000UL)                    /* System Control Space Base Address */
#define NVIC_BASE   (SCS_BASE +  0x0100UL)             /* NVIC Base Address */
#define NVIC        ((NVIC_Type *)NVIC_BASE)           /* NVIC configuration struct */
```

嵌套向量中断控制器 NVIC 的相关库函数定义在 misc.c 和 core_cm3.h 中，如表 6-10 所示。

表 6-10　NVIC 操作库函数

函　数　名	描　　述
NVIC_PriorityGroupConfig	设置优先级分组，抢占优先级和从优先级的位数
NVIC_Init	根据 NVIC_InitStruct 中指定的参数初始化外设 NVIC 寄存器
NVIC_SetVectorTable	设置向量表的位置和偏移
NVIC_SetPriorityGrouping	core_cm3 提供的设置中断优先级分组函数
NVIC_GetPriorityGrouping	core_cm3 提供的获取中断优先级分组函数
NVIC_EnableIRQ	使能一个外部中断
NVIC_DisableIRQ	清除一个外部中断
NVIC_GetPendingIRQ	获取某一外部中断的挂起状态
NVIC_SetPendingIRQ	设置某一外部中断的挂起状态
NVIC_ClearPendingIRQ	清除某一外部中断的挂起状态
NVIC_GetActive	获取某一外部中断的激活状态
NVIC_SetPriority	设置中断优先级(包括外部中断和系统异常)
NVIC_GetPriority	获取中断优先级(包括外部中断和系统异常)
NVIC_EncodePriority	根据优先级分组将抢占优先级和从优先级生成优先级
NVIC_DecodePriority	根据优先级分组将优先级分解为抢占优先级和从优先级

各个函数的具体功能如下：

1) void NVIC_PriorityGroupConfig(uint32_t NVIC_PriorityGroup)

该函数用于设置嵌套向量中断控制器的优先级分组。参数 NVIC_PriorityGroup 的取值为 0～4，分别对应中断优先级分组表中的各种分组情况。

```
#define NVIC_PriorityGroup_0    ((uint32_t)0x700)    //0 比特抢占优先级,4bit 从优先级
#define NVIC_PriorityGroup_1    ((uint32_t)0x600)    //1 比特抢占优先级,3bit 从优先级
#define NVIC_PriorityGroup_2    ((uint32_t)0x500)    //2 比特抢占优先级,2bit 从优先级
#define NVIC_PriorityGroup_3    ((uint32_t)0x400)    //3 比特抢占优先级,1bit 从优先级
#define NVIC_PriorityGroup_4    ((uint32_t)0x300)    //4 比特抢占优先级,0bit 从优先级
```

如果不执行该函数,默认的分组方式是分组 0,抢占优先级为 0,可以设置从优先级为 0～15 共 16 种情况。

2）void NVIC_Init(NVIC_InitTypeDef * NVIC_InitStruct)

该函数用于对嵌套向量中断控制器进行初始化。所有初始化信息都通过结构体指针 NVIC_InitStruct 进行传递,该指针指向 NVIC_InitTypeDef 类型。

NVIC_InitTypeDef 定义如下：

```
typedef struct
{
    uint8_t NVIC_IRQChannel;
    uint8_t NVIC_IRQChannelPreemptionPriority;
    uint8_t NVIC_IRQChannelSubPriority;
    FunctionalState NVIC_IRQChannelCmd;
} NVIC_InitTypeDef;
```

其中,NVIC_IRQChannel 设置中断通道,其取值由 stm32L1xx.h 中的 IRQn 枚举类型定义。

```
typedef enum IRQn
{
    NonMaskableInt_IRQn             = -14,
    MemoryManagement_IRQn           = -12,
    BusFault_IRQn                   = -11,
    UsageFault_IRQn                 = -10,
    SVC_IRQn                        = -5,
    DebugMonitor_IRQn               = -4,
    PendSV_IRQn                     = -2,
    SysTick_IRQn                    = -1,
    WWDG_IRQn                       = 0,
    PVD_IRQn                        = 1,
    TAMPER_STAMP_IRQn               = 2,
    RTC_WKUP_IRQn                   = 3,
    RCC_IRQn                        = 5,
    EXTI0_IRQn                      = 6,
    EXTI1_IRQn                      = 7,
    EXTI2_IRQn                      = 8,
    EXTI3_IRQn                      = 9,
    EXTI4_IRQn                      = 10,
    DMA1_Channel1_IRQn              = 11,
```

```
        DMA1_Channel2_IRQn              = 12,
        DMA1_Channel3_IRQn              = 13,
        DMA1_Channel4_IRQn              = 14,
        ...
} IRQn_Type;
```

NVIC_IRQChannelPreemptionPriority 设置抢占优先级，NVIC_IRQChannelSubPriority 设置从优先级，其取值范围均为 0～15；NVIC_IRQChannelCmd 设置 NVIC_IRQChannel 指定的中断通道使能，其取值为 ENABLE 和 DISABLE。

3）void NVIC_SetVectorTable(uint32_t NVIC_VectTab, uint32_t Offset)

该函数用于设置中断向量表的位置和偏移量，参数 NVIC_VectTab 用于指定中断向量表在 RAM 中还是 Flash 中，其取值为：NVIC_VectTab_RAM 和 NVIC_VectTab_FLASH；参数 Offset 用于指定中断向量表的偏移量，一般设定为 0。

4）void NVIC_SetPriorityGrouping(uint32_t PriorityGroup)

该函数由 core_cm3.h 定义，用于配置优先级分组，参数 ProrityGroup 的取值范围为 0～7，分别表示图 6-3 中 PRIGROUP 的八种抢占优先级和从优先级的分组情况。在 STM32L152 中，由于只使用了 4 个 bit 作为优先级配置，因此我们使用 misc.c 中提供 NVIC_PriorityGroupConfig 函数对优先级分组进行配置。

5）uint32_t NVIC_GetPriorityGrouping(void)

该函数用于读取 NVIC 控制器的优先级分组配置，其返回值为 0～7，含义见图 6-3 的 PRIGROUP 定义。

6）void NVIC_EnableIRQ(IRQn_Type IRQn)

该函数用于使能外部中断，输入参数 IRQn 为 0～239，即只能使能系统异常以外的外部中断，IRQn 的具体定义见 NVIC_Init 函数的 IRQn_Type 枚举类型定义。

7）void NVIC_DisableIRQ(IRQn_Type IRQn)

该函数用于屏蔽外部 IRQ 中断，输入参数 IRQn 为 0～239。

8）uint32_t NVIC_GetPendingIRQ(IRQn_Type IRQn)

该函数用于读取中断挂起寄存器并返回对应的 IRQn 编号的外部中断是否被挂起，若挂起，返回值为 1，否则返回值为 0. IRQn 必须为 0～239 的编号值。

9）void NVIC_SetPendingIRQ(IRQn_Type IRQn)

该函数用于配置挂起寄存器使得一个外部中断处于挂起状态，输入参数 IRQn 为要挂起的外部中断中断号，IRQn 的取值范围为 0～239。

10）NVIC_ClearPendingIRQ(IRQn_Type IRQn)

该函数用于清除一个外部中断的挂起位，输入参数 IRQn 的取值范围为 0～239。

11）uint32_t NVIC_GetActive(IRQn_Type IRQn)

该函数用于读取中断激活寄存器，返回输入参数 IRQn 的激活位，1 表示该外部中断处于激活状态，0 表示未激活，IRQn 的取值范围为 0～239。

12）void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)

该函数用于设置中断优先级，输入参数 IRQn 为中断号，取值范围为－14～239，即该函

数可以配置所有异常和中断的优先级,参数 priority 为所要设置的中断优先级,注意系统异常和外部中断的优先级值取值范围。

13) uint32_t NVIC_GetPriority(IRQn_Type IRQn)

该函数用于读取某个异常或中断的优先级,输入参数 IRQn 的取值范围为−14~239。

14) uint32_t NVIC_EncodePriority (uint32_t PriorityGroup, uint32_t PreemptPriority, uint32_t SubPriority)

该函数用于根据 PriorityGroup 参数指定的优先级分组,将抢占优先级 PreemptPriority 和从优先级 SubPriority 组合成一个 8bit 的优先级值。每种处理器实现 cortexM3 的优先级时可以不使用所有的 8bit(256 种优先级),其使用多少种优先级由__NVIC_PRIO_BITS 决定,在 STM32L152 中__NVIC_PRIO_BITS =4,即只支持 16 种优先级。如果该函数中的 PriorityGroup 值和之前设定的值不同,则以这两个值中的较小的值为准。

15) void NVIC_DecodePriority (uint32_t Priority, uint32_t PriorityGroup, uint32_t * pPreemptPriority, uint32_t * pSubPriority)

该函数用于根据输入的优先级 Priority 和优先级分组 PriorityGroup,将 Priority 分解成抢占优先级和从优先级。

【例 6-1】 中断控制器初始化案例。

```
void NVIC_Init(NVIC_InitTypeDef * NVIC_InitStruct)
{
  uint8_t tmppriority = 0x00, tmppre = 0x00, tmpsub = 0x0F;
  if (NVIC_InitStruct->NVIC_IRQChannelCmd != DISABLE)
  {  //计算优先级
    tmppriority = (0x700 - ((SCB->AIRCR) & (uint32_t)0x700))>>0x08;
    tmppre = (0x4 - tmppriority);
    tmpsub = tmpsub >> tmppriority;
    tmppriority = (uint32_t)NVIC_InitStruct->NVIC_IRQChannelPreemptionPriority << tmppre;
    tmppriority |= (uint8_t) (NVIC_InitStruct->NVIC_IRQChannelSubPriority & tmpsub);
    tmppriority = tmppriority << 0x04;
    NVIC->IP[NVIC_InitStruct->NVIC_IRQChannel] = tmppriority;
     //使能中断
    NVIC->ISER[NVIC_InitStruct->NVIC_IRQChannel >> 0x05] =
      (uint32_t)0x01 << (NVIC_InitStruct->NVIC_IRQChannel & (uint8_t)0x1F);
  }
  else
  { //清除中断使能位
    NVIC->ICER[NVIC_InitStruct->NVIC_IRQChannel >> 0x05] =
      (uint32_t)0x01 << (NVIC_InitStruct->NVIC_IRQChannel & (uint8_t)0x1F);
  }
}
```

【例 6-2】 采用库函数的 NVIC 中断配置实例如下所示。

```
void NVIC_Configuration(void)
{
  NVIC_InitTypeDef NVIC_InitStructure;
#ifdef  VECT_TAB_RAM
  /* 中断向量表位于 0x20000000 */
  NVIC_SetVectorTable(NVIC_VectTab_RAM, 0x0);
#else  /* VECT_TAB_FLASH  */
  /* 中断向量表位于 0x08000000 */
  NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x0);
#endif
  /* 中断优先级分组配置 */
  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
  /* 使能 EXTI9_5 中断 */
  NVIC_InitStructure.NVIC_IRQChannel = EXTI9_5_IRQn
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
}
```

【例 6-3】 采用库函数配置多个中断的实例如下所示。

```
/* 中断优先级分组配置 */
  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);
  /* 配置 TIM2 中断 */
  NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
  /* 配置 TIM3 中断 */
  NVIC_InitStructure.NVIC_IRQChannel = TIM3_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
  NVIC_Init(&NVIC_InitStructure);
  /* 配置 TIM4 中断 */
  NVIC_InitStructure.NVIC_IRQChannel = TIM4_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 2;
  NVIC_Init(&NVIC_InitStructure);
```

6.5 外部中断/事件控制器 EXTI

外部中断是由处理器的 I/O 引脚产生的中断，由于处理器的 I/O 引脚较多，使用也比

较灵活，为了便于管理这些 I/O 的中断，STM32L152 提供了一个外部中断/事件控制器专门处理外部中断。

外部中断/事件控制器由 23 个产生中断/事件请求的边沿检测器组成，其中 16 用于外部 I/O 引脚中断，7 个用于特殊事件处理，每个输入线可以独立地配置对应的触发事件，即上升沿、下降沿或双边沿触发。每个输入线都可以独立地被屏蔽。挂起位保持着中断请求的状态。外部中断/事件控制器的框图，如图 6-16 所示。

【思考题：中断和事件有何区别?】

由图 6-16 可以看出：

(1) 如果要产生中断，必须对中断线进行配置并使能该中断线。根据需要的边沿检测设置 2 个触发寄存器，同时在中断屏蔽寄存器相应位写 1 来允许中断请求。当外部中断线上发生了期待的边沿时，将产生一个中断请求，对应的挂起位被置为 1。

(2) 如果要产生事件，必须对事件线进行配置并使能该事件线。根据需要的边沿检测设置 2 个触发寄存器，同时在事件屏蔽寄存器相应位写 1 来允许事件请求。当事件线上发生了需要的边沿时，将产生一个事件请求脉冲，对应的挂起位不被置 1。

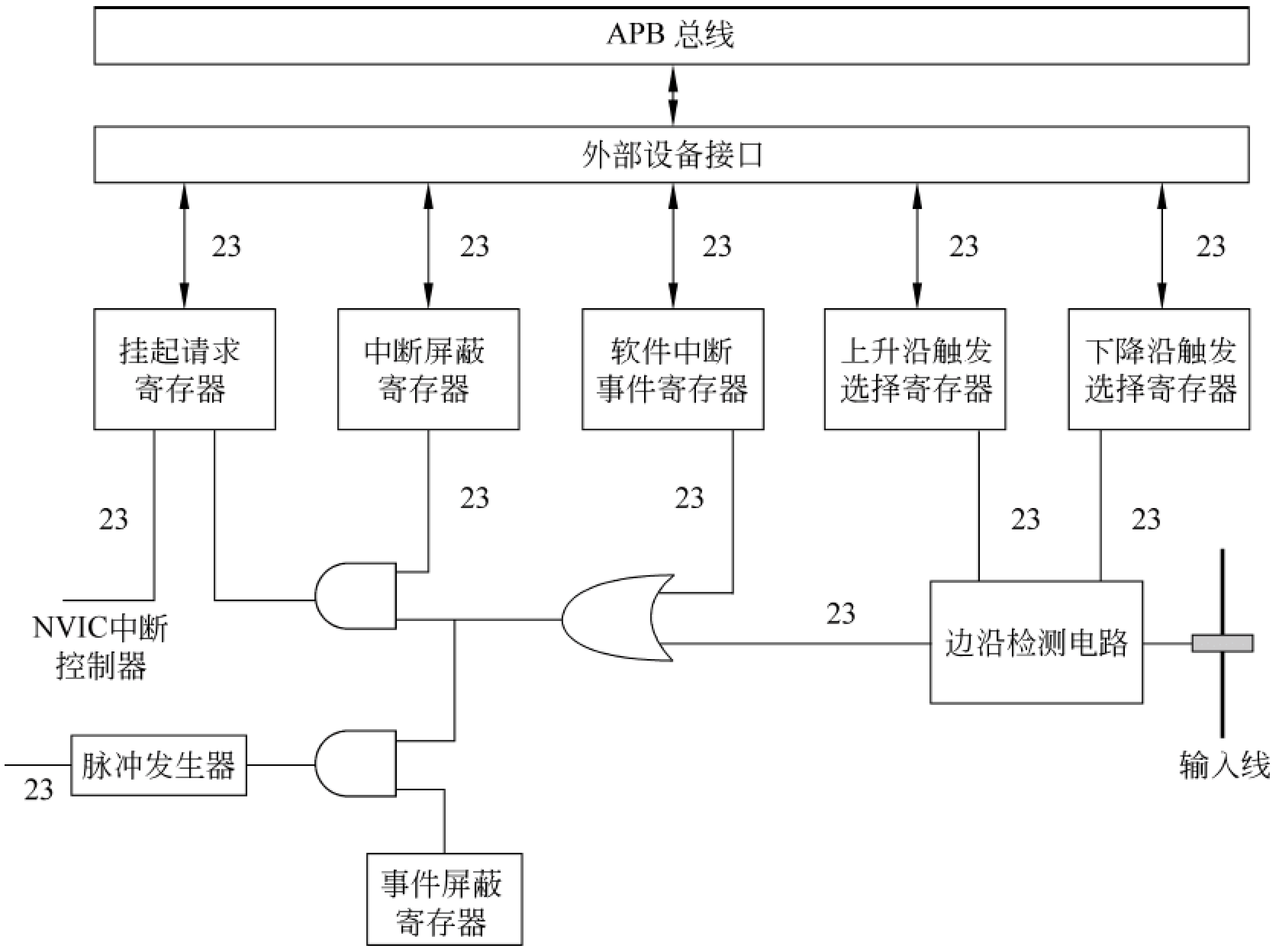


图 6-16　外部中断/事件控制器框图

(3) 通过软件向软件中断/事件寄存器写 1，也可以产生中断/事件请求。

EXTI 相关寄存器包括中断屏蔽寄存器(EXTI_IMR)、事件屏蔽寄存器(EXTI_EMR)、上升沿触发选择寄存器(EXTI_RTSR)、下降沿触发选择寄存器(EXTI_FTSR)、软件中断事件寄存器(EXTI_SWIER)、挂起寄存器(EXTI_PR)。

硬件中断配置过程：

(1) 配置中断线的屏蔽位(EXTI_IMR)；

(2) 配置所选中断线的触发选择位(EXTI_RTSR 和 EXTI_FTSR);

(3) 配置对应到外部中断控制器(EXTI)的 NVIC 中断通道的使能和屏蔽位,使得中断线的请求可以被正确地响应。

硬件事件配置过程:

(1) 配置事件线的屏蔽位(EXTI_EMR);

(2) 配置所选事件线的触发选择位(EXTI_RTSR 和 EXTI_FTSR)。

软件中断/事件配置过程:

(1) 配置中断/事件线的屏蔽位(EXTI_IMR 和 EXTI_EMR);

(2) 设置软件中断寄存器的请求位(EXTI_SWIER)。

6.6 寄存器说明

外部中断控制器的寄存器如表 6-11 所示。

表 6-11 EXTI 寄存器说明

寄存器名称	偏移量	功能	复位值
中断屏蔽寄存器 EXTI_IMR	0x00	设定中断是否屏蔽	0x 0000 0000
事件屏蔽寄存器 EXTI_EMR	0x04	设定事件是否屏蔽	0x 0000 0000
上升沿触发选择寄存器 EXTI_RTSR	0x08	配置中断/事件上升沿触发	0x 0000 0000
下降沿触发选择寄存器 EXTI_FTSR	0x0C	配置中断/事件下降沿触发	0x 0000 0000
软件触发中断寄存器 EXTI_SWIER	0x10	控制软件触发中断/事件	0x 0000 0000
挂起寄存器 EXTI_PR	0x14	中断的挂起状态	0x xxxx xxxx

1) 中断屏蔽寄存器(EXTI_IMR)

中断屏蔽寄存器是 32 位的寄存器,偏移地址为 0x00,复位值为 0x00000000,其有效位定义如图 6-17 所示。位 24~31 保留,必须始终保持为复位状态(0)。位 0~23 中位 x 的值为 0,则屏蔽来自线 x 的中断请求,位 0~23 中位 x 的值为 1,则开放来自线 x 的中断请求。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								MR23	MR22	MR21	MR20	MR19	MR18	MR17	MR16
								rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MR15	MR14	MR13	MR12	MR11	MR10	MR9	MR8	MR7	MR6	MR5	MR4	MR3	MR2	MR1	MR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 6-17 中断屏蔽寄存器

2) 事件屏蔽寄存器(EXTI_EMR)

事件屏蔽寄存器是 32 位的寄存器,偏移地址为 0x04,复位值为 0x00000000,其有效位

定义如图 6-18 所示。位 24～31 保留，必须始终保持为复位状态(0)。位 0～23 中位 x 的值为 0，则屏蔽来自线 x 的事件请求，位 0～23 中位 x 的值为 1，则开放来自线 x 的事件请求。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								MR23	MR22	MR21	MR20	MR19	MR18	MR17	MR16
								rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MR15	MR14	MR13	MR12	MR11	MR10	MR9	MR8	MR7	MR6	MR5	MR4	MR3	MR2	MR1	MR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 6-18　事件屏蔽寄存器

3）上升沿触发选择寄存器(EXTI_RTSR)

上升沿触发选择寄存器是 32 位的寄存器，偏移地址为 0x08，复位值为 0x00000000，其有效位定义如图 6-19 所示。位 24～31 保留，必须始终保持为复位状态(0)。位 0～23 中位 x 的值为 0，则禁止输入线 x 上的上升沿触发中断或事件，位 0～23 中位 x 的值为 1，则允许输入线 x 上的上升沿触发中断或事件。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								TR23	TR22	TR21	TR20	TR19	TR18	TR17	TR16
								rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 6-19　上升沿触发选择寄存器

4）下降沿触发选择寄存器(EXTI_FTSR)

下降触发选择寄存器是 32 位的寄存器，偏移地址为 0x0C，复位值为 0x00000000，其有效位定义如图 6-20 所示。位 24～31 保留，必须始终保持为复位状态(0)。位 0～23 中位 x 的值为 0，则禁止输入线 x 上下降沿触发中断或事件，位 0～23 中位 x 的值为 1，则允许输入线 x 上的下降沿触发中断或事件。

5）软件中断事件寄存器(EXTI_SWIER)

软件中断事件寄存器是 32 位的寄存器，偏移地址为 0x10，复位值为 0x00000000，其有效位定义如图 6-21 所示。位 24～31 保留，必须始终保持为复位状态(0)。位 0～23 中位 x 的值为 0，且相应的 EXTI_IMR 或 EXTI_EMR 寄存器中的中断位是 1(即开放来自线 x 的

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								TR23	TR22	TR21	TR20	TR19	TR18	TR17	TR16
								rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 6-20　下降沿触发选择寄存器

中断或事件),则写入 1 产生一个中断或事件请求。该位通过设置挂起寄存器 EXTI_PR 中的相应位 1 来清零。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								SWIER 23	SWIER 22	SWIER 21	SWIER 20	SWIER 19	SWIER 18	SWIER 17	SWIER 16
								rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SWIER 15	SWIER 14	SWIER 13	SWIER 12	SWIER 11	SWIER 10	SWIER 9	SWIER 8	SWIER 7	SWIER 6	SWIER 5	SWIER 4	SWIER 3	SWIER 2	SWIER 1	SWIER 0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 6-21　软件触发中断寄存器

6) 挂起寄存器(EXTI_PR)

软件中断事件寄存器是 32 位的寄存器,偏移地址为 0x14,复位值为 0x00000000,其有效位定义如图 6-22 所示。位 24～31 保留,必须始终保持为复位状态(0)。位 0～23 中位 x 的值为 0,则表明没有相应线的中断或事件请求产生。位 0～23 中位 x 的值为 1,则表明相应线的中断或事件请求已经产生。当向相应位写入 1,可以进行清零。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								PR23	PR22	PR21	PR20	PR19	PR18	PR17	PR16
								rw	rw	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PR15	PR14	PR13	PR12	PR11	PR10	PR9	PR8	PR7	PR6	PR5	PR4	PR3	PR2	PR1	PR0
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

图 6-22　挂起状态寄存器

由于 GPIO 引脚多,EXTI I/O 中断引脚少,因此需要对中断线和 I/O 进行映射,其映射关系如图 6-23 所示。

由图 6-23 可以看出,线 EXTIx(x=0,…,15)通过多路选择器来选择一个 GPIO 口 x 位作为中断。同一个时刻,不同端口的同一序号只能设置其中一个作为中断。16 个外部 I/O 中断均可以映射到不同的 I/O 引脚上,寄存器 SYSCFG_EXTICRn(n=1,2,3,4)用于 I/O 引脚的中断映射配置,每个 I/O 引脚占用 4 位,如 EXTI0 中断可配置到 PA0,PB0,PC0,…,PG0 八个 I/O 中的一个,寄存器有效位定义如图 6-24 所示。

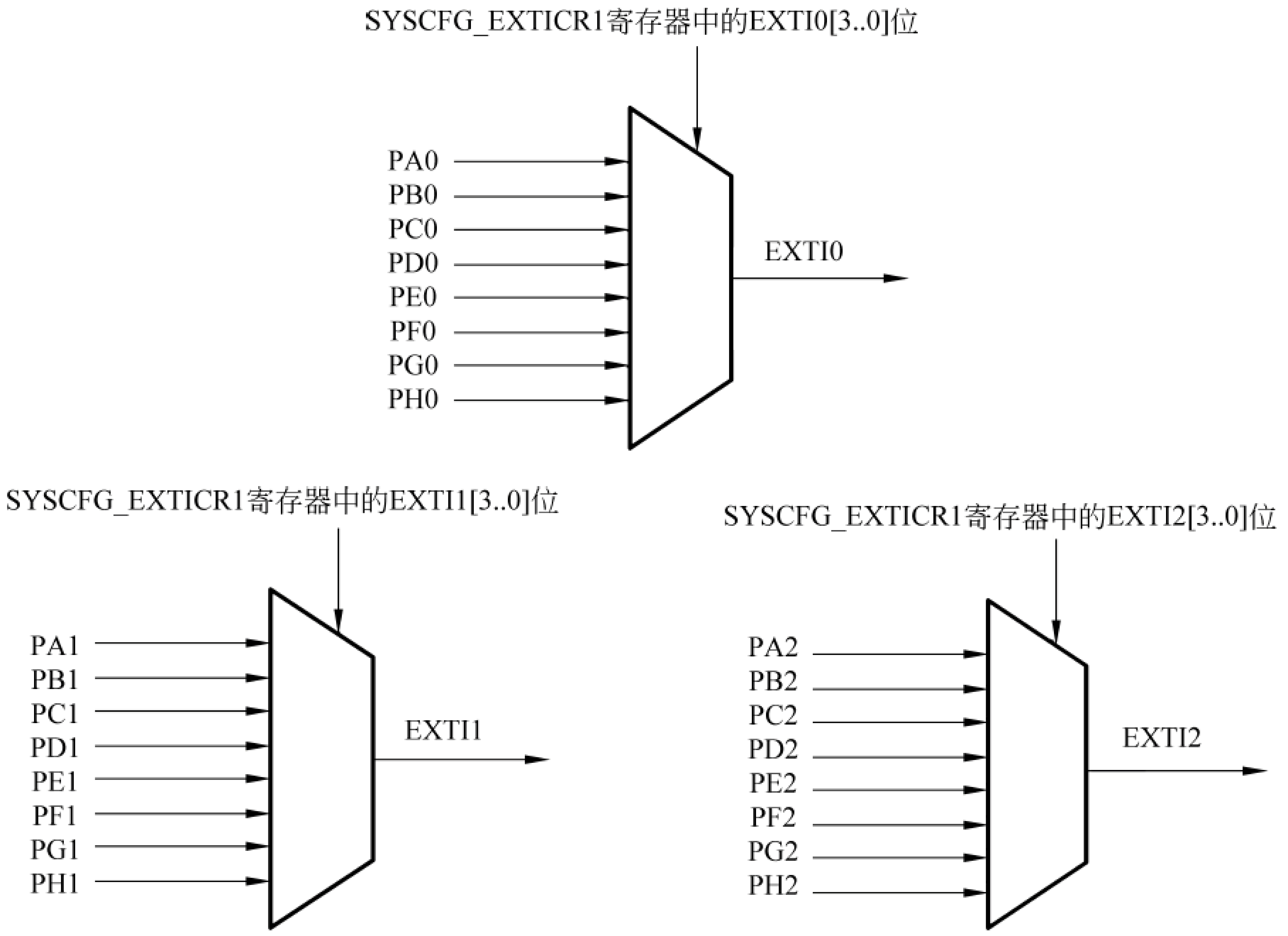


图 6-23　外部中断通用输入输出口映射

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXTI3[3:0]				EXTI2[3:0]				EXTI1[3:0]				EXTI0[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 6-24　外部中断源选择寄存器

SYSCFG_EXTICR2 用于配置 EXTI4～EXTI7，SYSCFG_EXTICR3 用于配置 EXTI8～EXTI11，SYSCFG_EXTICR4 用于配置 EXTI2～EXTI5。

其他的中断线分配如下：

(1) EXTI 中断线 16 连接到 PVD 输出；

(2) EXTI 中断线 17 连接到 RTC 闹钟事件；

(3) EXTI 中断线 18 连接到 USB 设备唤醒事件；

(4) EXTI 中断线 19 连接到 RTC Tamper 和 TimeStamp 事件；

(5) EXTI 中断线 20 连接到 RTC 唤醒事件；

(6) EXTI 中断线 21 连接到 Comparator1 唤醒事件；

(7) EXTI 中断线 22 连接到 Comparator2 唤醒事件；

(8) EXTI 中断线 23 连接到通道获取中断。

6.7 EXTI 函数库

EXTI 寄存器结构 EXTI_TypeDef 在文件 stm32L1xx.h 中定义。

```
typedef struct
{
    __IO uint32_t IMR;          /* 外部中断屏蔽寄存器 */
    __IO uint32_t EMR;          /* 外部事件屏蔽寄存器 */
    __IO uint32_t RTSR;         /* 外部中断/事件上升沿配置寄存器 */
    __IO uint32_t FTSR;         /* 外部中断/事件下降沿配置寄存器 */
    __IO uint32_t SWIER;        /* 软件中断/事件配置寄存器 */
    __IO uint32_t PR;           /* 中断/事件挂起寄存器 */
} EXTI_TypeDef;
```

外部中断的库函数如表 6-12 所示。

表 6-12　外部中断 EXTI 的相关的库函数

函　数　名	功　　能
EXTI_DeInit	将外设 EXTI 寄存器设为默认值
EXTI_Init	用 EXTI_InitStruct 中指定的参数初始化外设 EXTI 寄存器
EXTI_StructInit	将 EXTI_InitStruct 中的每一个参数按默认值填入
EXTI_GenerateSWInterrupt	产生一个软中断
EXTI_GetFlagStatus	检查指定的 EXTI 挂起寄存器
EXTI_ClearFlag	清除 EXTI 挂起寄存器
EXTI_GetITStatus	检查指定的 EXTI 线路触发请求发生与否
EXTI_ClearITPendingBit	清除 EXTI 线路挂起位

1. 函数 EXTI_Init

功能描述：根据 EXTI_InitStruct 中指定的参数初始化外设 EXTI 寄存器。

函数原型：void EXTI_Init(EXTI_InitTypeDef * EXTI_InitStruct)。

输入参数 EXTI_InitStruct：指向结构 EXTI_InitTypeDef 的指针，包含了外设 EXTI 的配置信息，EXTI_InitTypeDef 的定义如下：

```
typedef struct
{
    uint32_t EXTI_Line;
    EXTIMode_TypeDef    EXTI_Mode;
    EXTITrigger_TypeDef    EXTI_Trigger;
```

```
    FunctionalState   EXTI_LineCmd;
} EXTI_InitTypeDef;
```

EXTI_Line 选择待配置的外部中断中断线，其取值为 EXTI_Linex，其中 x 为 0～15。

EXTI_Mode 设置待配置中断线的中断模式，其取值为：

- EXTI_Mode_Event：设置 EXTI 中断线为事件请求；
- EXTI_Mode_Interrupt：设置 EXTI 中断线为中断请求。

EXTI_Trigger 设置待配置中断线的边沿触发模式，其取值为：

- EXTI_Trigger_Falling：设置输入中断线为下降沿触发；
- EXTI_Trigger_Rising：设置输入中断线为上升沿触发；
- EXTI_Trigger_Rising_Falling：设置输入中断线为下降沿和上升沿触发。

EXTI_LineCmd 用来定义待配置中断线的使能状态，其取值为 ENABLE 或者 DISABLE。

例如使能外部中断 12 和 14、下降沿触发：

```
EXTI_InitTypeDef EXTI_InitStructure;
EXTI_InitStructure.EXTI_Line = EXTI_Line12 | EXTI_Line14;
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;
EXTI_InitStructure.EXTI_LineCmd = ENABLE;
EXTI_Init(&EXTI_InitStructure);
```

其中，EXIT_Line 设置初始化的外部中断线；EXTI_Mode 设置外部中断模式，取值为 EXTI_Mode_Interrupt 表示产生中断，取值为 EXTI_Mode_Event 表示产生事件；EXTI_Trigger 设置外部中断触发方式，取值为 EXTI_Trigger_Rising 表示上升沿触发，取值为 EXTI_Trigger_Falling 表示下降沿触发，取值为 EXTI_Trigger_Rising_Falling 表示双沿触发；EXTI_LineCmd 设置外部中断线使能。

2. 函数 EXTI_GetFlagStatus

功能描述：检查指定的 EXTI 中断线是否挂起。

函数原型：FlagStatus EXTI_GetFlagStatus(uint32_t EXTI_Line)。

输入参数 EXTI_Line，待读取的 EXTI 中断线名称。

返回值：EXTI_Line 的状态（SET 或者 RESET）。

例如：

```
FlagStatus EXTIStatus;
EXTIStatus = EXTI_GetFlagStatus(EXTI_Line8);
```

3. 函数 EXTI_ClearFlag

功能描述：清除 EXTI 中断线挂起标志位。

函数原型：void EXTI_ClearFlag(uint32_t EXTI_Line)。

输入参数 EXTI_Line，待清除的 EXTI 中断线。

例：

```
EXTI_ClearFlag(EXTI_Line2);
```

4. 函数 EXTI_GetITStatus

功能描述：检查指定的外部中断线上的触发请求是否发生。

函数原型：ITStatus EXTI_GetITStatus(uint32_t EXTI_Line)。

输入参数 EXTI_Line：待检查 EXTI 线路。

返回值：EXTI_Line 的状态(SET 或者 RESET)。

例：

```
EXTIStatus = EXTI_GetITStatus(EXTI_Line8);
```

5. 函数 EXTI_ClearITPendingBit

功能描述：用于清除指定的外部中断线上的中断挂起位，使得在中断发生后进入中断服务程序时，可以继续响应中断请求。

函数原型：void EXTI_ClearITPendingBit(uint32_t EXTI_Line)。

输入参数 EXTI_Line：待清除 EXTI 中断线。

例：

```
EXTI_ClearITpendingBit(EXTI_Line2);
```

6. 函数名 SYSCFG_EXTILineConfig

功能描述：选择 GPIO 引脚作为中断输入，该函数不属于 EXTI 控制器库函数。

函数原型：SYSCFG_EXTILineConfig(uint8_t EXTI_PortSourceGPIOx, uint8_t EXTI_PinSourcex)。

输入参数 1：GPIO 引脚 EXTI_PortSourceGPIOx，x 的取值范围为 A～H。

输入参数 2：配置的中断通 EXTI_PinSourcex，x 取值范围为 0～15。

例如，将 EXTI0 连接到 PA0 引脚：

```
SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOA, EXTI_PinSource0);
```

6.8 中断案例

使用 I/O 口外部中断的一般步骤为：

(1) 初始化 I/O 口为输入。

(2) 开启 I/O 口复用时钟，设置 I/O 口与中断线的映射关系。

(3) 初始化线上中断，设置触发条件等。

(4) 配置中断分组(NVIC)，并使能中断。

（5）编写中断服务函数。

【例 6-4】　外部中断配置方法：

```
EXTI_InitTypeDef EXTI_InitStructure;
EXTI_InitStructure.EXTI_Line=EXTI_Line4;                     //中断线的标号 EXTI_Line0~EXTI_Line15
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;          //中断模式:事件、中断
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;      //触发方式
EXTI_InitStructure.EXTI_LineCmd = ENABLE;
EXTI_Init(&EXTI_InitStructure);
//涉及中断要设置 NVIC 中断优先级
NVIC_InitTypeDef NVIC_InitStructure;
NVIC_InitStructure.NVIC_IRQChannel = EXTI2_IRQn;             //使能按键外部中断通道
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x02; //抢占优先级 2
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x02;        //子优先级 2
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;              //使能外部中断通道
NVIC_Init(&NVIC_InitStructure);
```

在 NVIC 初始化前，我们还需要调用 NVIC_PriorityGroupConfig(NVIC_PriorityGroup_x)配置优先级分组。

配置完中断优先级之后，接着我们要做的就是编写中断服务函数。中断服务函数的名字是在.s 的汇编文件中事先有定义。STM32 的 I/O 口外部中断函数只有 6 个，分别为：

```
EXTI0_IRQHandler
EXTI1_IRQHandler
EXTI2_IRQHandler
EXTI3_IRQHandler
EXTI4_IRQHandler
EXTI9_5_IRQHandler
EXTI15_10_IRQHandler
```

即中断线 0～4 每个中断线对应一个中断函数，中断线 5～9 共用中断函数 EXTI9_5_IRQHandler，中断线 10～15 共用中断函数 EXTI15_10_IRQHandler。

在编写中断服务函数的时候会经常使用到两个函数：

（1）判断某个中断线上的中断是否发生(标志位是否置位)：

ITStatus EXTI_GetITStatus(uint32_t EXTI_Line)；这个函数一般使用在中断服务函数的开头判断中断是否发生。

（2）清除某个中断线上的中断标志位：

void EXTI_ClearITPendingBit(uint32_t EXTI_Line)；这个函数一般应用在中断服务函数结束之前，清除中断标志位。

常用的中断服务函数格式为：

```
void EXTI2_IRQHandler(void)
{
```

```
    if(EXTI_GetITStatus(EXTI_Linex) != RESET)                  //判断 LINEx 线上的中断是否发生
    {
        中断逻辑…
        EXTI_ClearITPendingBit(EXTI_Line3);                    //清除 LINEx 上的中断标志位
    }
}
```

【例 6-5】 通过连接到 PA0 口的按键产生中断,来切换连接到 PB7 口的 LED 灯显示。

```
#include "stm32l1xx.h"
#include "stm32l1xx_gpio.h"
int main(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;                     //通用输入输出的初始化的结构体
    EXTI_InitTypeDef EXTI_InitStructure;                     //外部中断控制器的初始化的结构体
    NVIC_InitTypeDef NVIC_InitStructure;                     //嵌套向量中断控制器的初始化的结构体
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOB, ENABLE); //使能 GPIOB
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7;             //设置 PB7,用于连接 LED 灯
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;         //设置为输出模式
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;        //设置输出模式下的推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;     //设置最大速度为 40MHz
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;      //设置无上拉
    GPIO_Init(GPIOB, &GPIO_InitStructure);                //初始化 GPIOB
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE); //使能 GPIOA
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG,ENABLE);//使能 SYSCFG
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;             //设置 PA0,用于产生按键
                                                          //输入
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;          //设置为输入模式
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;      //设置无上拉
    GPIO_Init(GPIOA, &GPIO_InitStructure);                //初始化 GPIOA
    //连接 EXTI 线和 GPIO 引脚(根据输入输出口外部中断映射)
    SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOA,EXTI_PinSource0);  //EXTI 线 0
    EXTI_InitStructure.EXTI_Line = EXTI_Line0;
    EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;        //配置外部中断模式
    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;     //上升沿触发方式
    EXTI_InitStructure.EXTI_LineCmd = ENABLE;                  //使能外部中断线 0
    EXTI_Init(&EXTI_InitStructure);                            //初始化外部中断控制
                                                               //器 EXTI
    NVIC_InitStructure.NVIC_IRQChannel = EXTI0_IRQn;           //NVIC 中断源
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x0F; //抢占优先级 15
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x0F;      //设置从优先级 15
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;            //使能中断通道
    NVIC_Init(&NVIC_InitStructure);                            //初始化嵌套向量中断控
                                                               //制器
```

```
    while (1);
}
void Delay(__IO uint32_t nCount)
{
  while(nCount--);
}
```

文件 stm32l1xx_it.c 中的中断服务程序，代码如下：

```
void EXTI0_IRQHandler(void)
{
  //检测外部中断线 0 上的触发请求
  if(EXTI_GetITStatus(EXTI_Line0) != RESET)
  {
    //切换 LED1 状态
    GPIO_ToggleBits(GPIOB, GPIO_Pin_7);
    //清除 EXTI 线 0 上的挂起位
    EXTI_ClearITPendingBit(EXTI_Line0);
  }
}
```

【例 6-6】 全局开关中断。

按键按下 1 次，关中断，再次按下，开中断，代码如下：

```
while (1)
{   //判断按键是否按下弹起
    while(GPIO_ReadInputDataBit(GPIOA,GPIO_Pin_0) ==RESET);
    while(GPIO_ReadInputDataBit(GPIOA,GPIO_Pin_0) != RESET);
    __disable_irq();                                    //关闭全局中断
    GPIO_SetBits(GPIOB,GPIO_Pin_7);                     //亮灯
    //判断第二次按键是否按下弹起
    while(GPIO_ReadInputDataBit(GPIOA,GPIO_Pin_0) =RESET);
    while(GPIO_ReadInputDataBit(GPIOA,GPIO_Pin_0) != RESET);
    __enable_irq();                                     //开全局中断
    GPIO_ResetBits(GPIOB,GPIO_Pin_7);                   //灭灯
}
```

Chapter 7

第7章　定　时　器

【导读】 定时器是嵌入式系统中使用最频繁的部件，本章首先介绍定时器（Timer）的基本组成和工作原理，然后对Cortex-M3内部定时器和STM32L152外围定时器分别作了介绍。内部定时器SysTick功能简单，适合驱动操作系统，方便不同厂家微控制器之间的移植；外围定时器包括基本定时器和通用定时器，通用定时器功能较为复杂，除了定时外，还具有输入捕获、输出比较、PWM产生等功能，多个硬件定时器之间还可以进行级联。STM32L152的8个外围定时器的寄存器有所区别，本章对寄存器定义统一进行了描述，对特殊部分进行了标注，介绍了常用寄存器的各个域的功能，以及CMSIS提供的典型寄存器操作库函数，最后以定时中断、比较输出、输入捕获和PWM为例介绍了定时器使用方法。

7.1　定时器原理概述

定时器是嵌入式系统中最为常用的一个功能模块，定时器为应用系统提供延迟、定时中断、捕获输入、控制PWM输出等一系列功能，也是驱动操作系统运行的关键硬件模块。

我们之前介绍GPIO时使用了Delay函数，通过一个for循环来实现延时。Delay函数的执行需要CPU参与，即CPU在Delay期间处于运行状态且不能执行其他程序，对于嵌入式系统，这种实现功耗较高，系统性能受到影响。定时器是一个独立于CPU的硬件，采用中断方式和CPU进行交互，可以有效提高嵌入式微控制器整体性能。定时器的基本原理是通过一个计数器自动计数，当计数到某个特定值时，触发一个中断，计数的周期和计数值用于确定计数器持续的时间。

定时器的基本原理如图7-1所示，由预分频器、自动重装载寄存器、计数器、比较寄存器和中断输出单元构成。

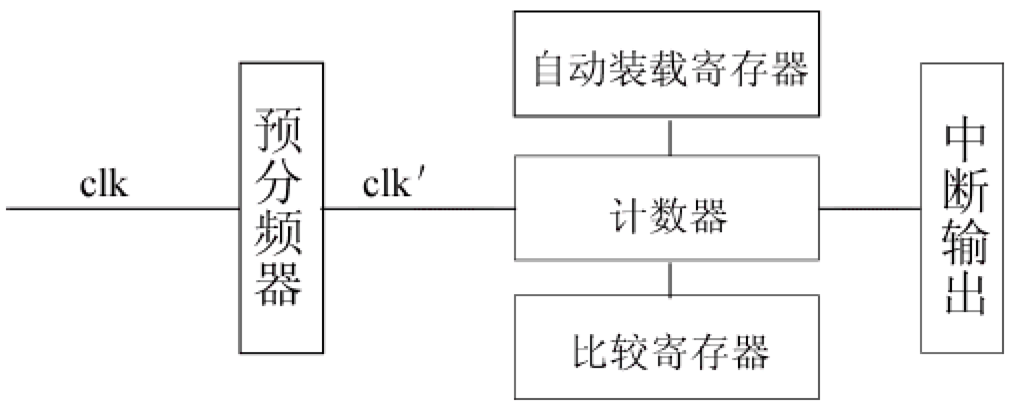


图7-1　定时器基本结构

计数器在时钟驱动下工作，每个时钟脉冲计数器自动加 1 或减 1，时钟的频率由预分频器决定，预分频器可以将定时器的输入时钟进行 1，2，4，8 等分频，这样一个计数值可以表示更长的时间间隔。自动重装载寄存器是用户可配置的，计数器在自加模式下，从 0 开始计数，当计数值和自动重装载寄存器值相同时，产生一个中断，并将计数器值清零；当计数器工作在自减模式时，计数器的初值为自动重装载寄存器的值，当计数器递减到 0 时产生一个中断，并将计数器的初值重新赋为自动重装载寄存器值。如果没有配置自动重装载寄存器，则计数器默认的装载值为计数器能表示的最大值（溢出值）。除了计数器时间到产生中断外，有些定时器还支持比较寄存器中断输出方式，即当计数器的值和比较寄存器值相同时产生一个中断，用户可以不断设置比较寄存器的值在计数器溢出前产生多次中断。

定时器的计时长度等于计数值/预分频后的时钟频率，一般定时器的计数器为 8bit、16bit 和 32bit，我们称为 8 位定时器、16 位定时器和 32 位定时器。定时器位数越长，计时长度越长；预分频越小，一个时间脉冲的长度越短，表达的时间越精确，但定时器能表示的最长时间间隔越短。因此在使用定时器时，需要根据需求调整预分频参数和定时器长度。

在嵌入式系统中，我们需要控制多个时间值或者控制不同精度的定时，一般嵌入式微控制器会提供多个硬件定时器，这些定时器可以并行使用，但软件系统用到的定时器根据应用不同可能会有很多，因此硬件定时器不够时我们需要基于一个硬件定时器实现多个软件定时功能，以满足系统需求。

【思考题：如何通过一个硬件定时器实现 n 个软件定时器？**】**

为适应不同嵌入式系统的需求，STM32L152 提供了丰富的定时器功能，结构上也比图 7-1 所示的定时器基本结构复杂，主要包括 SysTick 定时器、通用定时器和基本定时器（不同于 STM32F 系列，L 系列没有高级定时器）。此外，看门狗定时器和实时时钟 RTC 也可作为普通定时使用。

SysTick 是一个 24 位的倒计数定时器，当计到 0 时，将自动装载定时初值，其主要用于为操作系统提供一个硬件的滴答中断，进行进程调度。SysTick 定时器由 Cortex-M3 定义，存在于 NVIC 控制器中，这样便于同是 Cortex-M3 内核、不同厂家的嵌入式处理器进行系统移植。在无操作系统的嵌入式系统中，该计时器可以作为一个普通定时器使用。

STM32L152 外围控制器提供了通用定时器和基本定时器，两类定时器的功能不同，通用定时器除了基本定时器功能外，还有向上/向下计数、PWM、输出比较、输入捕获等功能，不同的通用定时器在捕获通道数量、编码器等功能上也有所区别。STM32L152 提供的定时器比较如表 7-1 所示。

表 7-1　STM32L152 外围定时器比较

定　时　器	计数模式	计数器长度	捕获/比较通道数量	其 他 功 能
通用定时器 TIM2、TIM3、TIM4	向上、向下、向上/向下	16 位	4	外部时钟触发和定时器同步，支持编码器
通用定时器 TIM9	向上	16 位	2	外部时钟触发和定时器同步

续表

定　时　器	计数模式	计数器长度	捕获/比较通道数量	其 他 功 能
通用定时器 TIM10、TIM11	向上	16 位	1	无
基本定时器 TIM6、TIM7	向上	16 位	0	DAC 触发

7.2　内部定时器 SysTick

SysTick 定时器是 Cortex-M3 内核自带的定时器，该定时器的时钟源一般采用内核时钟，因此，采用 SysTick 使得软件在不同的 Cortex-M3 处理器上的移植更加方便。由于内核时钟频率较高，SysTick 定时器具有较高精度，通常可用于精确计时和测量。

SysTick 定时器在 NVIC 中，Cortex-M3 为该定时器设有 SYSTICK 异常。SysTick 是一个 24 位倒计时定时器，最大可计数 2^{24}，计数值保存在 STK_VAL 寄存器中，每过一个时钟周期，STK_VAL 的值减 1，当减到 0 时，触发 SYSTICK 异常，同时硬件自动把重装载寄存器 STK_LOAD 中的数据加载到 STK_VAL，重新开始向下计数。

7.2.1　SysTick 寄存器

SysTick 定时器的控制涉及四个寄存器，如表 7-2 所示。

表 7-2　SysTick 定时器寄存器及其功能

寄存器名	寄存器地址	读 写 权 限	功　　能
SYST_CSR	0xE000E010	RW	SysTick 控制和状态寄存器
SYST_RVR	0xE000E014	RW	SysTick 重装载寄存器
SYST_CVR	0xE000E018	RW	SysTick 计数值寄存器
SYST_CALIB	0xE000E01C	RO	SysTick 校准寄存器

SysTick 定时器的主要寄存器的具体定义如下。

1）SysTick 控制和状态寄存器 SYST_CSR

SYST_CSR 是一个 32 位寄存器，如图 7-2 所示，其有效域有四个。

其中 bit[16]为 COUNTFLAG 计数标志位，用来表示 SysTick 的计数值是否已经数到了 0，如果已经到 0，则该位被置 1，如果读取该位，该位将被自动清零。

bit[2]为 CLKSOURCE，表明 SysTick 时钟源，该位为 1 时表示采用处理器主时钟 HCLK 作为时钟源，0 表示采用 HCLK/8 作为时钟源。

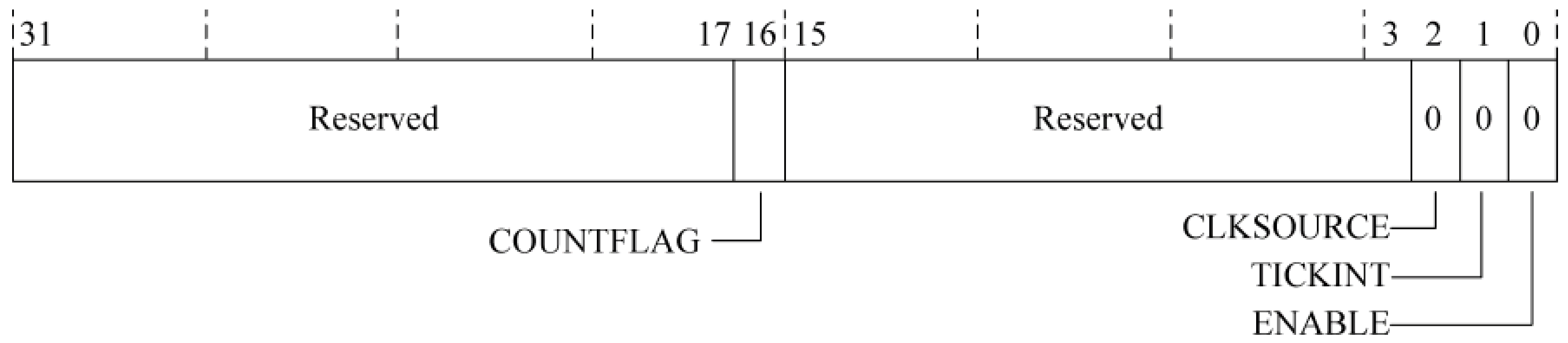


图 7-2　SysTick 控制和状态寄存器

Bit[1]为 TICKINT 中断使能位，该位为 1 表示 SYST_CSR 计数到 0 时产生中断，0 表示不产生中断。该位为 0 时，可以通过监测 COUNTFLAG 判断 SYST_CSR 是否计数到 0。

Bit[0]为 ENABLE 使能位，该位为 1 表示启用 SysTick 定时器，0 表示关闭 SysTick 定时器。

2）SysTick 重载寄存器 SYST_RVR

SYST_RVR 是一个 32 为寄存器，如图 7-3 所示，有效位为低 24 位 bits[23：0]，保存 SYST_CVR 计数到 0 时加载到 SYST_CVR 寄存器的计数值。RELOAD 的取值范围为 0x00000001～0x00FFFFFF，要产生 N 个时钟周期的中断，RELOAD 设为 N－1。

图 7-3　SysTick 重载寄存器

3）SysTick 当前计数值寄存器 SYST_CVR

SYST_CVR 中存放 SysTick 定时器当前的计数值，如图 7-4 所示有效位 24 位，读取该寄存器时返回 CURRENT 的值，写该寄存器，CURRENT 将被置 0，同时 SYST_CSR 的 COUNTFLAG 位也将被置 0。

图 7-4　SysTick 计数值寄存器

4）SysTick 校准值寄存器 SYST_CALIB

SYST_CALIB 用于存放 SysTick 定时器校准值，即 4M 时钟 1ms 时间间隔的计数值，默认为 4000。

SysTick 定时器的配置流程和工作过程如下：

- 配置 SYST_RVR，设定定时器的计数周期数。
- 清空 SYST_CVR 寄存器的计数值。
- 配置 SYST_CSR 寄存器，设定 SysTick 计数器的时钟源，是否启用中断。
- 配置 SYST_CSR 寄存器，使能 ENABLE，启动定时器。

当 ENABLE 为 1 时,定时器将 SYST_RVR 寄存器的 RELOAD 值加载到 SYST_CVR 并开始向下计数,当计数到 0 时,将 SYST_CSR 寄存器的 COUNTFLAG 位置 1 并根据 TICKINT 的值确定是否产生中断请求,然后将 SYST_RVR 寄存器的 RELOAD 值再次加载到 SYST_CVR 寄存器,启动向下计数。

如果允许中断,调用中断处理程序中;如不启用中断,可通过不断读取 SYST_CSR 寄存器的 COUNTFLAG 标志位判断是否计时至零。

7.2.2 SysTick 定时器库函数

为了便于对 SysTick 定时器进行操作,CMSIS 提供了 SysTick 定时器的寄存器定义和库函数。

SysTick 定时器的寄存器结构体定义在 core_cm3.h 头文件中,我们可以通过结构体类型 SysTick_Type 定义变量对 SysTick 定时器进行控制。

```
typedef struct {
__IO uint32_t CTRL;                        //SysTick 控制和状态寄存器
__IO uint32_t LOAD;                        //SysTick 重载寄存器
__IO uint32_t VAL;                         //SysTick 计数值寄存器
__IO uint32_t CALIB;                       //SysTick 校准值寄存器
} SysTick_Type;
```

【例 7-1】 用 SysTick 定时器产生任意大小 T 的精确延迟,以微秒为单位。

分析:我们首先选取 SysTick 定时器的时钟源,HCLK 或 HCLK/8,STM32L152 的最高频率 HCLK 为 32MHz,因此以 HCLK 为时钟源,一个时钟周期为 1/32 微秒,因此对于任意大小时间 T,其重载寄存器的值应为 32 * T。定义一个 SysTick_Type 类型的变量配置 SysTick 定时器的寄存器,通过查询 CTRL 寄存器的 COUNTFLAG 判断计时是否到,代码如下:

```
SysTick_Type SysTick;                      //定义一个 SysTick类型的定时器变量
void Delay_us(uint32_t n)                  //单位为 μs
{
  SysTick->LOAD=32*n;                      //配置重载寄存器值
  SysTick->CTRL=0x00000005;                //时钟源 HCLK(32M),打开定时器
  while(!(SysTick->CTRL&0x00010000));      //等待计数到 0
  SysTick->CTRL=0x00000004;                //关闭定时器
}
```

CMSIS 为 SysTick 定时器提供了两个操作函数,如表 7-3 所示,可直接调用方便使用,其中 SysTick_CLKSourceConfig 定义在 misc.c 中,SysTick_Config 定义在 core_cm3.h 中。

表 7-3　SysTick 库函数

函 数 名 称	函 数 功 能
SysTick_CLKSourceConfig	配置 SysTick 定时器时钟源
SysTick_Config	初始化并开启 SysTick 计数器及中断

1）SysTick_CLKSourceConfig 函数

函数原型：void SysTick_CLKSourceConfig(uint32_t SysTick_CLKSource)。

输入参数：SysTick_CLKSource，用于表明 SysTick 定时器的时钟源，其取值为：

- SysTick_CLKSource_HCLK_Div8，HCLK/8 作为 SysTick 时钟源；
- SysTick_CLKSource_HCLK，HCLK 作为 SysTick 时钟源；

示例：

```
SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK);
//设置 AHB 时钟为 SysTick 时钟源,32MHz
```

2）SysTick_Config 函数

函数原型：uint32_t SysTick_Config(uint32_t ticks)。

输入参数：ticks，两次中断间的间隔数值。

函数返回值，0 表示成功，1 表示失败。

SysTick_Config()函数将参数 ticks 写到 Systick 的重载寄存器中，配置 SysTick 中断优先级为最低(0x0F)，清除 Systick 当前计数值寄存器，配置时钟源为 HCLK，使能中断并启动计数。定时器时间(秒)＝ticks/HCLK。具体实现如下：

```
uint32_t SysTick_Config(uint32_t ticks)
{
  //检查 ticks,如果 ticks 超过 24 位,返回失败
  if (ticks > SysTick_LOAD_RELOAD_Msk)   return (1);
  //配置装载寄存器,SysTick_LOAD_RELOAD_Msk= 0xFFFFFFFFul
  SysTick->LOAD  = (ticks & SysTick_LOAD_RELOAD_Msk) - 1;
  //配置 SYSTICK 中断优先级,设为最低 0xF
  NVIC_SetPriority(SysTick_IRQn, (1<<__NVIC_PRIO_BITS) - 1);
  SysTick->VAL = 0;                          //计数值清零
  //配置控制寄存器,CLKSOURCE= HCLK,中断使能,定时器启动
  SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk | SysTick_CTRL_TICKINT_Msk
                  | SysTick_CTRL_ENABLE_Msk;
  return (0);                                //初始化成功返回 0
}
```

中断发生后，调用 SysTick 的中断处理程序 SysTick_Handler，在文件 stm32l1xx_it.c，SysTick_Handler 的定义如下：

```
void SysTick_Handler(void)                   //systick 中断处理函数
```

```
{
}
```

如果需要更改时钟源，在 SysTick_Config 函数后调用 SysTick_CLKSource Config(SysTick_CLKSource_HCLK_Div8)；如果需要更改 SysTick 中断优先级，在 SysTick_Config 函数后调用 NVIC_SetPriority(SysTick_IRQn,. number.)。

示例：

```
SysTick_Config(320)                          //设置 SysTick定时器时间为 10μs
```

7.2.3 SysTick 定时器应用例程

【例 7-2】 启动用 SysTick 定时器中断，实现 LED 灯控制。

分析：对例 5-1 的 LED 控制程序进行修改，利用 SysTick_Config 配置 SysTick 定时器，在 SysTick 中断函数中设置状态变量 flag 进行翻转，main 函数中根据 flag 对灯进行控制，代码如下：

```
#include "stm32l1xx.h"
#include "stm32L1xx_gpio.h"
int flag=0;
GPIO_InitTypeDef          GPIO_InitStructure;
int main(void)
{
   //配置 PB7 为推挽输出模式
   RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOB, ENABLE);
   GPIO_InitStructure.GPIO_Pin=GREEN_LED;
   GPIO_InitStructure.GPIO_Mode=GPIO_Mode_OUT;
   GPIO_InitStructure.GPIO_OType=GPIO_OType_PP;
   GPIO_InitStructure.GPIO_Speed=GPIO_Speed_40MHz;
   GPIO_InitStructure.GPIO_PuPd=GPIO_PuPd_NOPULL;
   GPIO_Init(GPIOB, &GPIO_InitStructure);
   SysTick_Config(500* 32* 1000)             //配置定时器 500ms
   while (1)
   {
      if(flag)                               //根据 flag对灯进行控制
         GPIO_SetBits(GPIOB, GPIO_Pin_7);
      else
         GPIO_ResetBits(GPIOB, GPIO_Pin_7);
   }
}
```

在文件 stm32l1xx_it.c 中的 SysTick_Handler 中断服务程序中，添加如下代码：

```
extern int flag;
void SysTick_Handler(void)                        //SysTick中断处理函数
{
  if(flag)
      flag = 0;
  else
    flag = 1;
}
```

7.3　外围定时器基本概念

除了 Cortex-M3 内核带的 SysTick 定时器外，STM32L152 还在外围总线上提供了多个硬件定时器，这些硬件定时器命名为 TIMx(x=2,3,4,6,7,9,10,11)。外围定时器 TIMx 比 SysTick 定时器功能更为强大，相比图 7-1，结构也较为复杂，由时基单元和捕获比较单元组成。

1. 时基单元

时基单元是定时器的基本单元，包括预分频器、计时器和自动装载寄存器，完成最基本的计时功能。时基单元带有一个自动重装载的累加计数器，计数器的时钟通过一个预分频器得到。软件可以读写计数器、自动重装载寄存器和预分频寄存器，计数器运行时也可以进行读写操作。

时基单元的主要寄存器包括：

- 计数器寄存器(TIMx_CNT)。
- 预分频寄存器(TIMx_PSC)。
- 自动重装载寄存器(TIMx_ARR)。

定时器时基单元的配置与定时器驱动时钟、计数方式的选择有密切关系，以下对 STM32L152 的定时器时钟源和计数方式进行简要介绍。

1) 时钟源

定时器的工作时钟来源于预分频器分频后的时钟 CK_CNT，预分频器的输入时钟为 CK_PSC，CK_PSC 的时钟来源包括以下四种，每个 TIMx 支持的时钟源也不同：

- 内部时钟(CK_INT)。
- 外部时钟模式 1：外部输入脚(TIx)。
- 外部时钟模式 2：外部触发输入(ETR)。
- 内部定时器触发输入(ITRx)。

(1) 内部时钟源(CK_INT)：即定时器采用 CK_INT 作为时钟源，CK_INT 由微控制器的总线 APB1 或 APB2 的时钟提供，定时器的时钟不是直接来自 APB1 或 APB2，而是来自于 APB1 或 APB2 的一个倍频器，当 APB1 或 APB2 的预分频系数为 1 时，这个倍频器不

起作用,定时器的时钟频率等于 APB1 或 APB2 的频率;当 APB1 或 APB2 的预分频系数为其他数值(2、4、8 或 16)时,定时器的时钟频率等于 APB1 或 APB2 的频率两倍。如图 7-5 所示,当 APB1 总线被设置为 32MHz 时(预分频为 1),CK_INT 的时钟为 32MHz,当 APB1 总线被设置为 16MHz 时,此时倍频器起作用,CK_INT 的时钟频率被设为 32MHz。

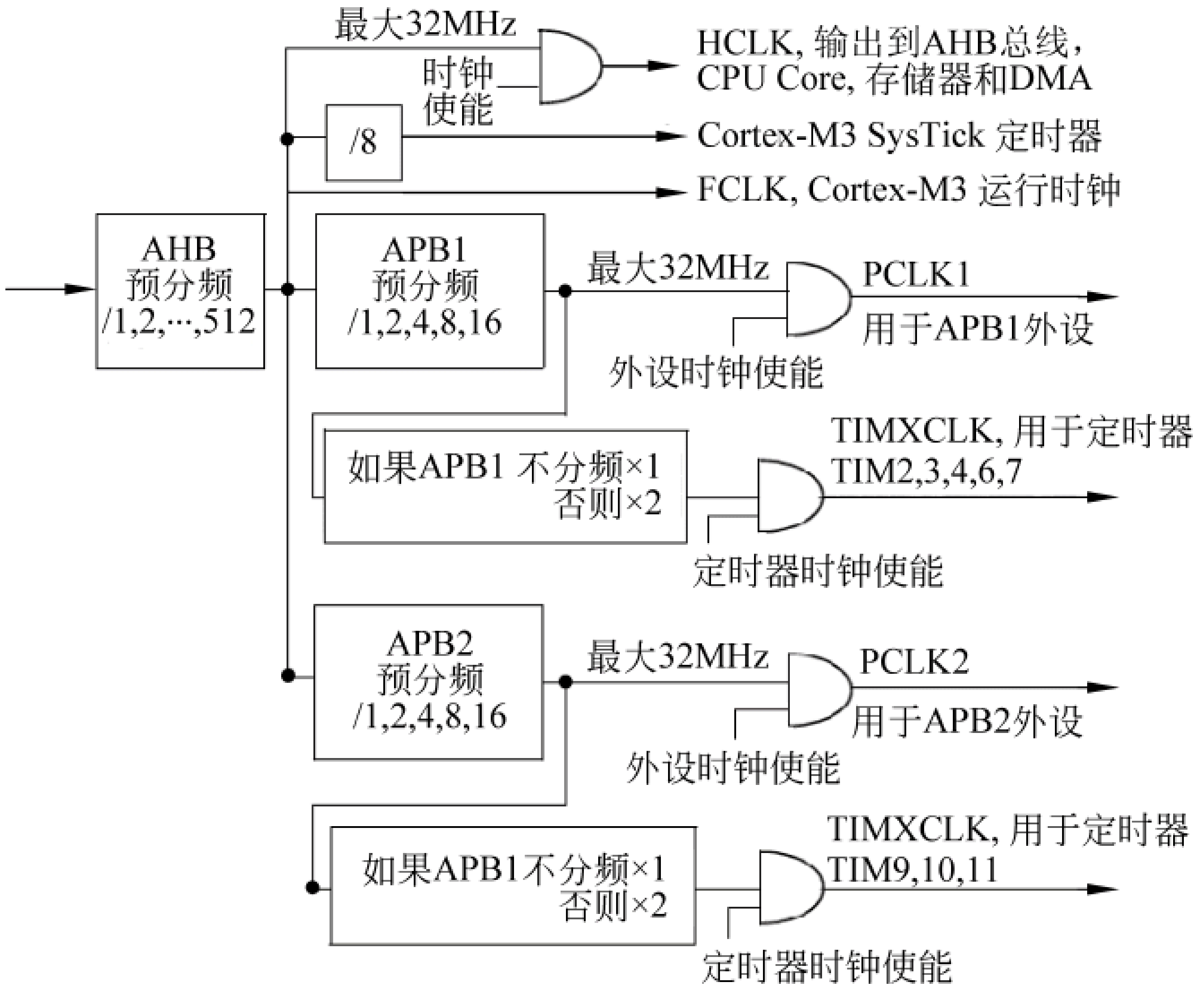


图 7-5　定时器内部时钟源

STM32L152 中,TIM2～TIM7 连接在 APB1 总线,TIM9～TIM11 连接在 APB2 总线,因此在使用不同的 TIMx 时需要配置不同的总线频率。

(2) 外部时钟源模式 1 下,时钟来源于 MCU 外部引脚,时钟源选择 TIMx 输入通道 1 或输入通道 2 所对应引脚,定时器在 TIMx 输入通道对应引脚的电平边沿信号作为时钟驱动。

(3) 外部时钟源模式 2 下,时钟源来源于 MCU 外部引脚,时钟源为外部触发引脚 ETR,定时器在 ETR 引脚的每一个上升沿或下降沿计数。

(4) 内部定时器触发模式下,可以使用一个定时器作为另一个定时器的预分频器时钟输入,每个定时器最多可以有 4 个其他内部定时器触发时钟源,如 TIM4 可以用 TIM10、TIM2、TIM3 和 TIM9 作为内部时钟源,具体参见 STM32L152 参考手册。

基本定时器 TIM6 和 TIM7 的时钟只能由内部时钟提供,但通用定时器 TIM2～TIM5、TIM9～TIM11 可以选择多种时钟源。目前定时器使用中,基本上都是采用内部时钟作为时钟源。

2) 计数方式

计数模式包括三种模式:向上计数模式、向下计数模式和中央对齐模式。

(1) 向上计数模式:计数器从 0 计数到自动加载值(TIMx_ARR 计数器的值),然后重新从 0 开始计数并且产生一个计数器溢出事件和更新事件,当发生一个更新事件时,所有的

寄存器都被更新，如图 7-6 所示。

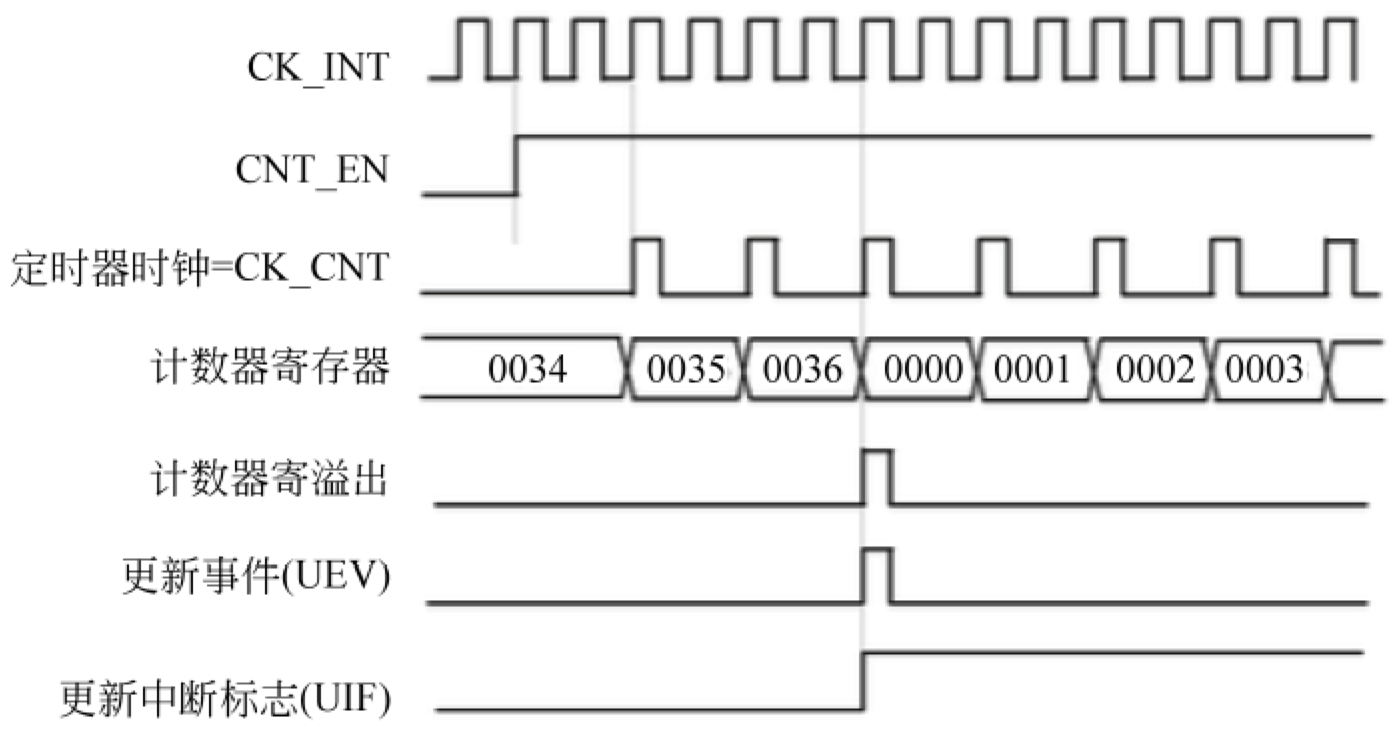


图 7-6　向上计数模式

(2) 向下计数模式：在向下模式中，计数器从自动装入的值(TIMx_ARR 计数器的值)开始向下计数到 0，然后从自动装入的值重新开始，并产生一个计数器向下溢出事件和更新事件，如图 7-7 所示。

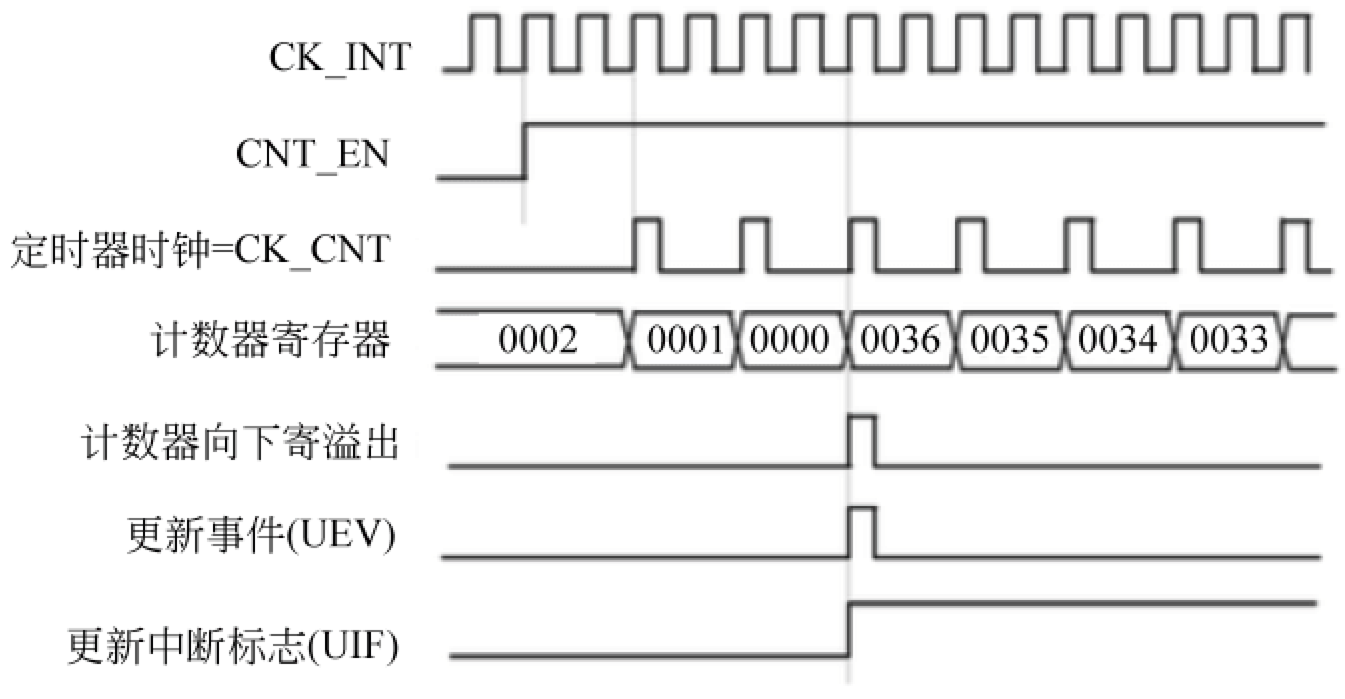


图 7-7　向下计数模式

(3) 中央对齐模式：在中央对齐模式，计数器从 0 开始计数到自动加载值−1，即 TIMx_ARR−1，产生一个计数器上溢事件，然后向下计数到 1 并且产生一个计数器下溢事件，然后再从 0 开始重新计数。计数的方向由硬件寄存器指示，可以在每次计数上溢和每次计数下溢时产生更新事件，然后，计数器重新从 0 开始计数，如图 7-8 所示。

2. 捕获比较单元

捕获比较单元可对输入信号进行捕捉，或根据设定输出不同的信号。

(1) 输入捕获：可以用来捕获外部事件，并为其赋予时间标记以说明此事件的发生时刻。外部事件发生的触发信号由单片机中对应的外部引脚输入，比如按下按键，也可以通过内部单元(如模拟比较器等)来实现。捕获的信号可以设置为上升沿捕获、下降沿捕获、或者上升沿下降沿都捕获，当设置的捕获发生时，微控制器会将计数寄存器的值复制到捕获比较寄存器，当再次捕捉到电平变化时，新的捕获比较寄存器的值减去之前复制的值就是输入信

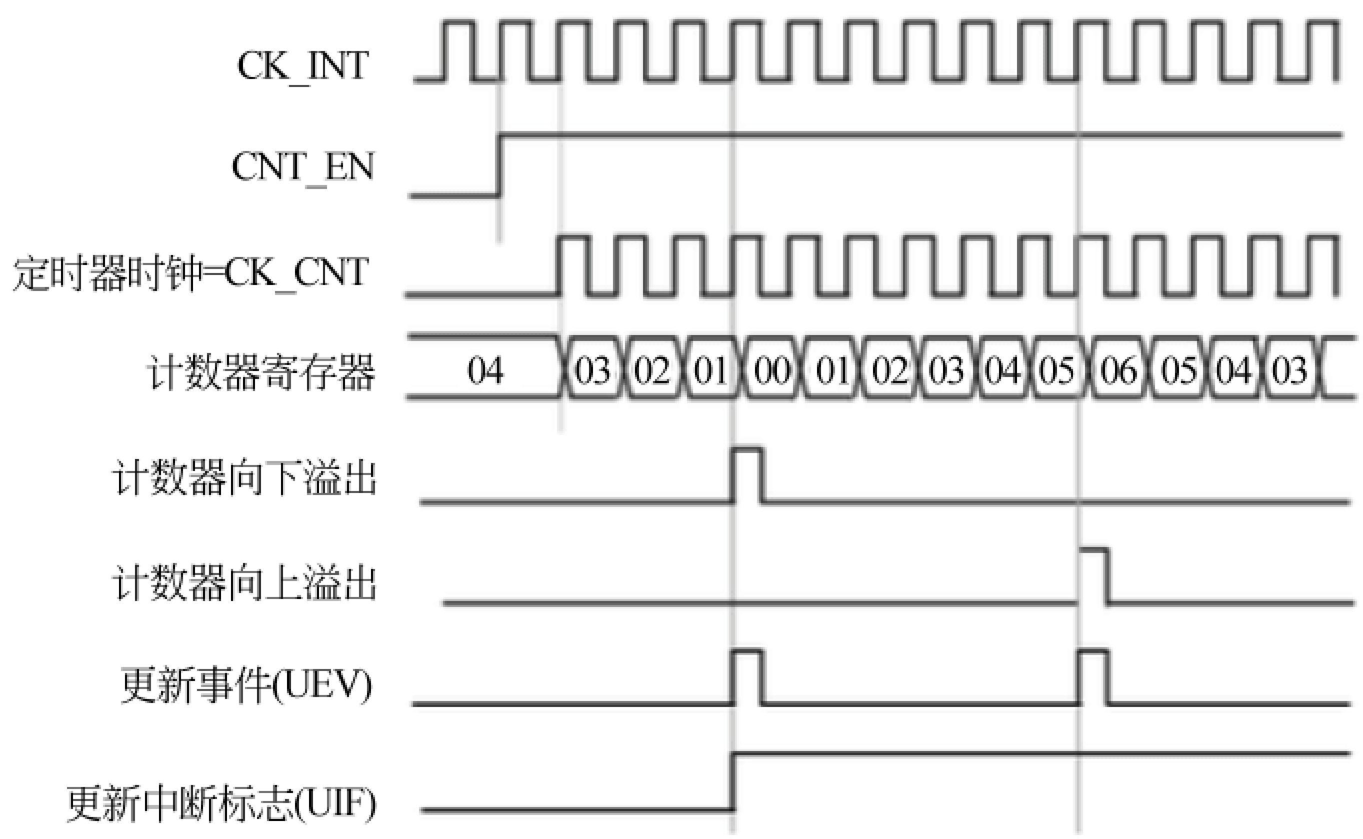


图 7-8　中央计数模式

号的间隔或持续时间。

(2) 输出比较：定时器中计数寄存器在初始化完后会自动的计数，一旦计数寄存器在计数过程中与比较寄存器匹配则会产生匹配事件，此时我们可以产生一个中断，也可以在 GPIO 的端口输出一个电平变化(变低、变高或取反)，用于控制其他外设。

STM32L152 的不同定时器支持的捕获和输出比较通道的数量不同，两通道捕获和输出比较电路原理如图 7-9 所示，GPIO 端口 TIMx_CH1 和 TIMx_CH2 可作为捕获输入源或者比较输出源使用。作为输入捕获时，首先对输入信号进行滤波，生成 TI1F，然后进行上升沿、下降沿检测，生成信号 TI1FP1，选通器可以选择通道 1，也可以选择通道 2 作为捕获信号 IC1，即通道 1 同时连接到了 IC1 和 IC2，这样连接的目的可以用两个捕获通道对于同一个信号的不同边沿进行检测。分频器可根据配置对 IC1 进行分频，分频后的 IC1PS 脉冲边沿被监测到时，将计数器的值保存到捕获比较寄存器，并产生捕获事件。作为比较输出时，捕获比较寄存器与计数器的值进行比较，当发生匹配时，产生信号 OC1REF，通过输出控制在 TIMx_CH1 端口输出高电平或低电平。

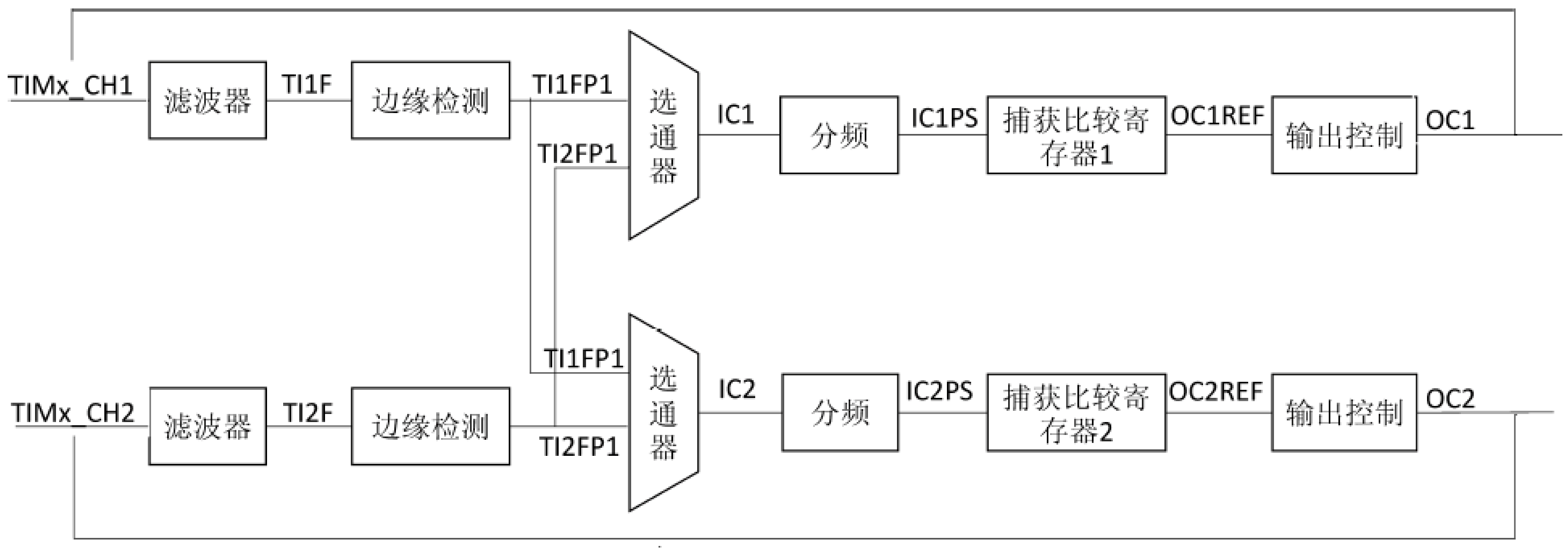


图 7-9　捕获和输出比较原理

3. 定时器工作中的中断和事件

定时器工作中，会在一些关键点触发一些事件，用于管理定时器的状态，比如计数器向上计数模式中，计数器从 0 计数到用户定义的比较值(TIMx_ARR 寄存器的值)，然后重新从 0 开始计数并产生一个计数器溢出事件。事件发生时，定时器的相关寄存器会记录事件状态，但并不一定会向 CPU 发起中断请求，只有中断允许被配置的情况下定时器才会发生中断请求。因此定时器事件和中断是不同的概念，定时器中断依赖于定时器事件，但定时器事件发生并不一定产生中断。为了便于软件对硬件定时器的管理，定时器支持软件产生事件的功能，当读写定时器状态寄存器的相关控制域时，可以立即生成一个事件，便于程序控制定时器，也可以通过控制寄存器禁用事件产生。

4. 影子寄存器

在定时器控制中，一些寄存器在电路实现时通常对应有两个寄存器，一个用于用户读写访问，称之为预加载寄存器，一个是用户看不见但用于实际控制，称之为影子寄存器。例如，时基单元的自动重装载寄存器带有影子寄存器，用户对于自动重载寄存器的读写实际上是通过读写预加载寄存器实现的。这样的好处是当定时器正在工作时，读写自动重装载寄存器不会影响原先定时器的工作，所有真正需要起作用的寄存器可以在同一个特定条件(如更新事件产生)触发时才把预加载寄存器的值写入到影子寄存器，以保证多个通道控制的同步性，当然用户也可配置成立即写入影子寄存器的方式。本章以下部分定时器结构图中，凡是带有阴影的寄存器都有影子寄存器。

7.4　基本定时器 TIM6、TIM7

基本定时器包括两个独立的 16 位定时器 TIM6 和 TIM7，可用于一般的定时时钟或作为驱动模数转换的 DAC 输出时钟(内部已连接到 DAC)，其定时器结构如图 7-10 所示。

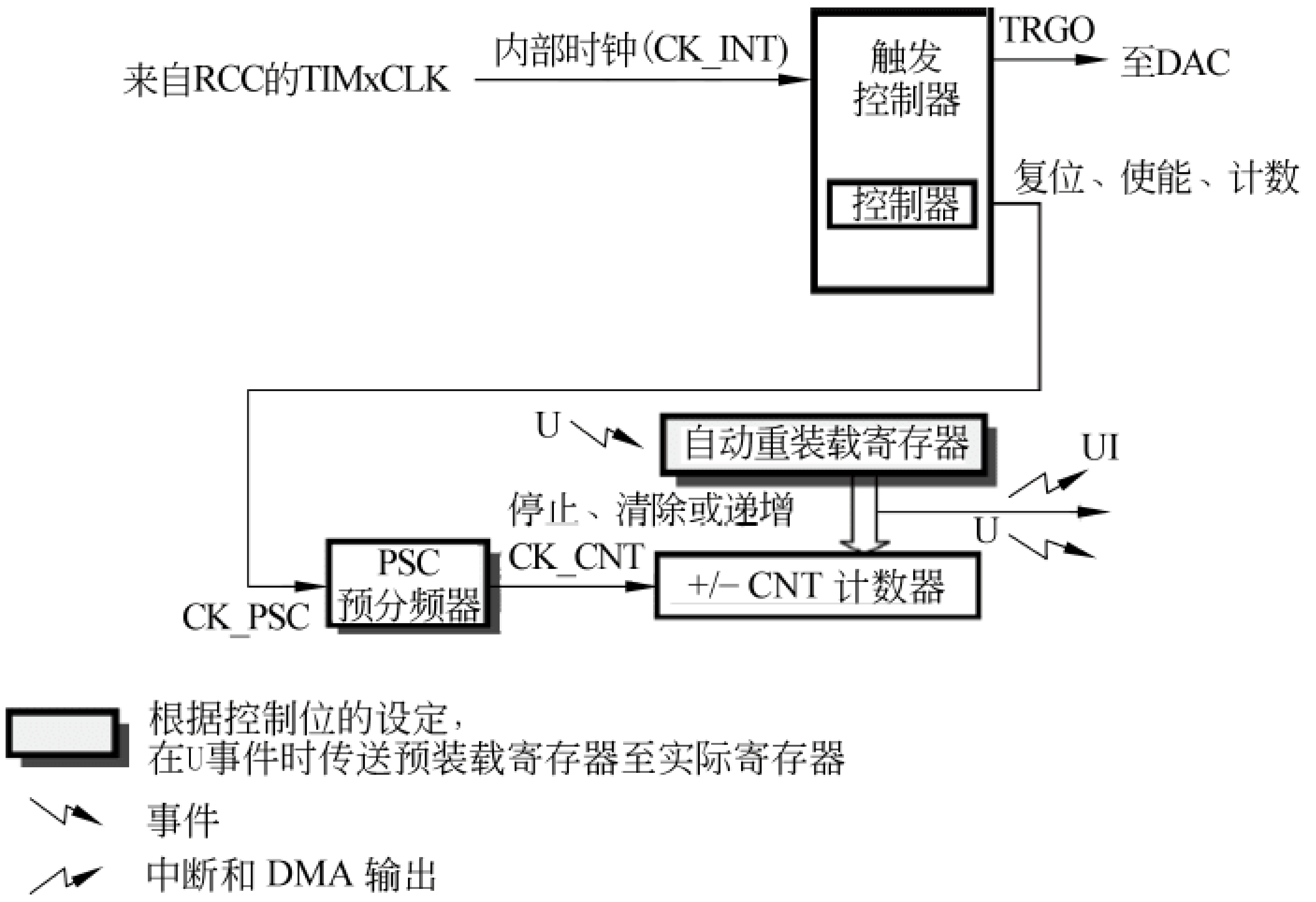


图 7-10　基本定时器结构

基本定时器由控制单元和时基单元组成。控制单元用于配置定时器的参数，获取定时器工作状态，主要由控制寄存器 TIMx_CR1、TIMx_CR2，中断使能寄存器 TIMx_DIER、状态寄存器 TIMx_SR 和事件产生寄存器 TIMx_EGR 组成，在更新事件发生时可以产生中断/DMA 请求。时基单元用于定时器计数控制，主要包括计数寄存器 TIMx_CNT、预分频寄存器 TIMx_PSC 和 自动重装载寄存器 TIMx_ARR。

基本定时器连接在 APB1 总线上，其时钟由内部时钟 CK_INT 提供，CK_INT 不超过总线的最高频率 32MHz，为满足定时器的精度和计时时间长短的要求，可以通过预分频器对总线时钟进行分频。计数器由预分频输出的 CK_CNT 驱动，预分频器是通过一个 16 位寄存器（TIMx_PSC）实现分频，可以以 1～65536 的任意数值对计数器时钟分频。

如图 7-10 所示，TIMx_PSC 预分频寄存器和 TIMx_ARR 带有影子寄存器，在运行过程读写中改变 TIMx_PSC 和 TIMx_ARR 的数值实际改变的是预加载寄存器的值，预加载寄存器的值将在下一个更新事件 UEV 时写入到影子寄存器。TIMx_CR1 寄存器中的自动重装载预加载使能位 ARPE 决定是将写入预加载寄存器的内容立即传送到影子寄存器，还是在更新事件 UEV 时传送到影子寄存器。

如图 7-11 和图 7-12 所示，发生一次更新事件 UEV 时，定时器将设置更新标志位，并将以下寄存器立即更新：

- 传送 TIMx_PSC 预装载值至预分频器的影子寄存器；
- 更新自动重装载影子寄存器为 TIMx_ARR 预装载值。

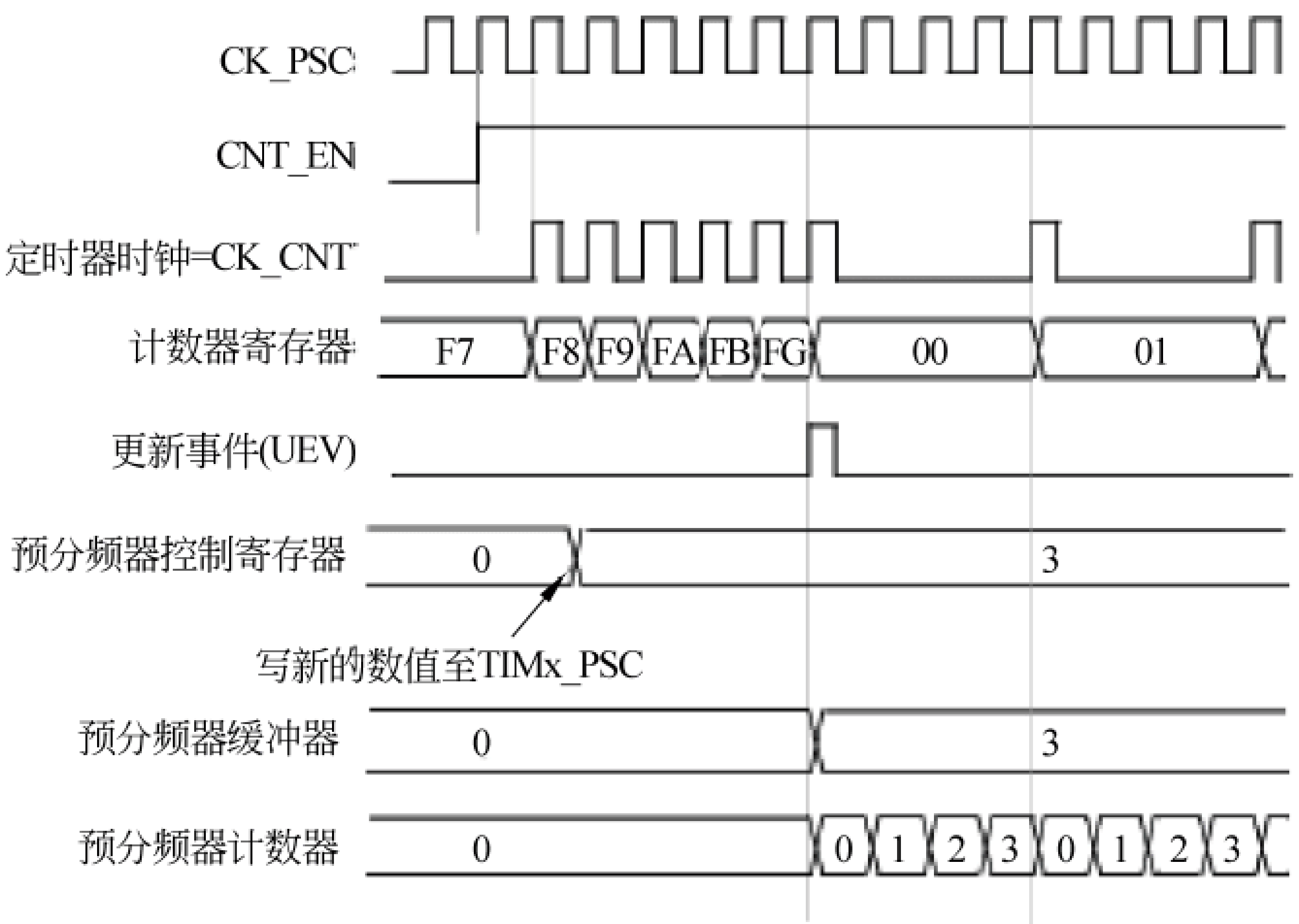


图 7-11 PSC 预装值操作示例

基本定时器的操作流程和工作过程如下：

- 配置预分频寄存器 TIMx_PSC，计数器的时钟频率 CK_CNT 等于 fCK_PSC/（PSC[15：0]＋1）；
- 配置自动重装载寄存器 TIMx_ARR 的计数值；

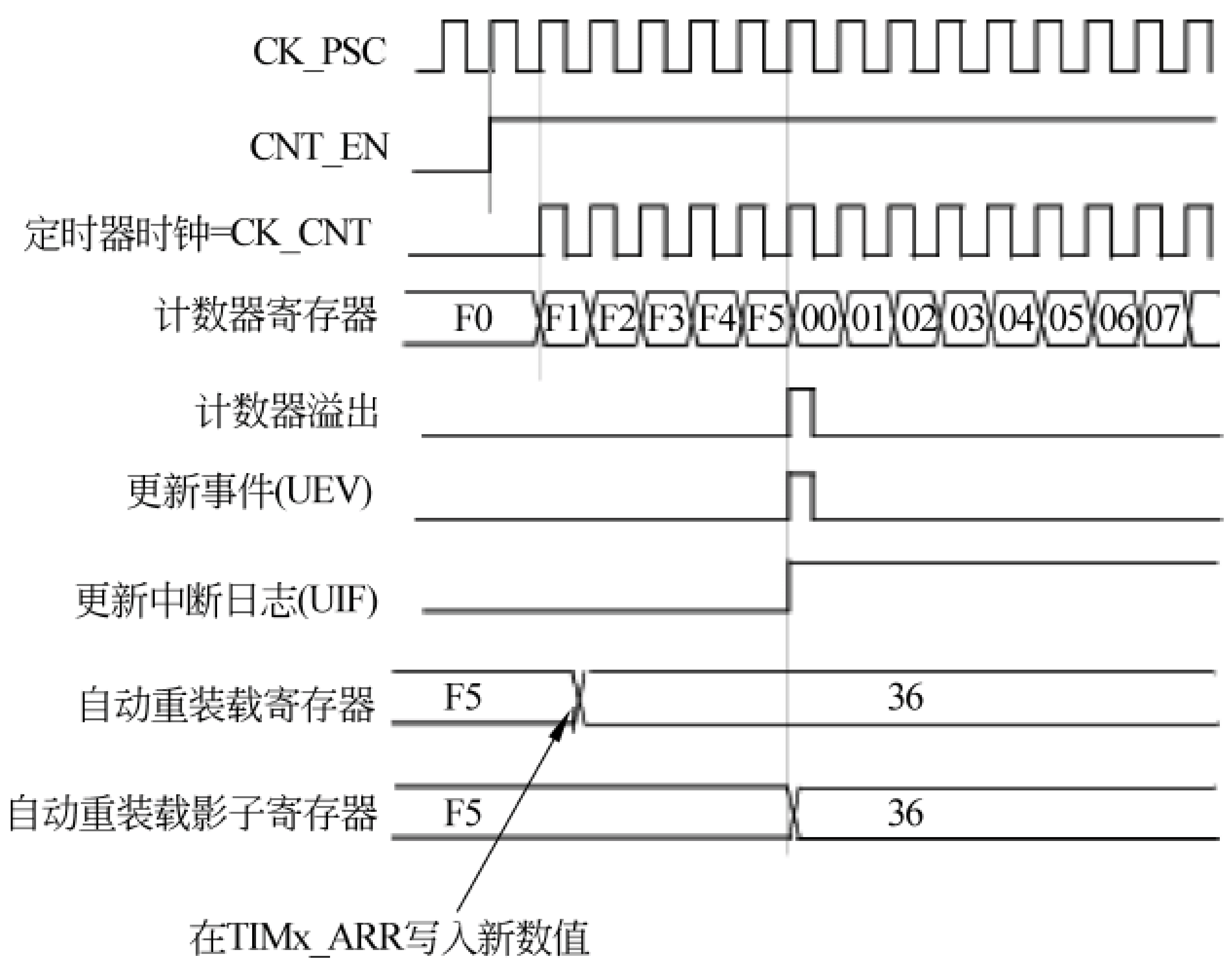


图 7-12　ARR 预装载操作示例

- 配置中断使能寄存器 TIMx_DIER，是否启动 DMA 和中断；
- 配置事件产生寄存器 TIMx_EGR 清空计数器，产生一个软件更新事件，更新所有寄存器配置；
- 配置控制寄存器 TIMx_CR1，设置 CEN 位为 1，启动定时器。

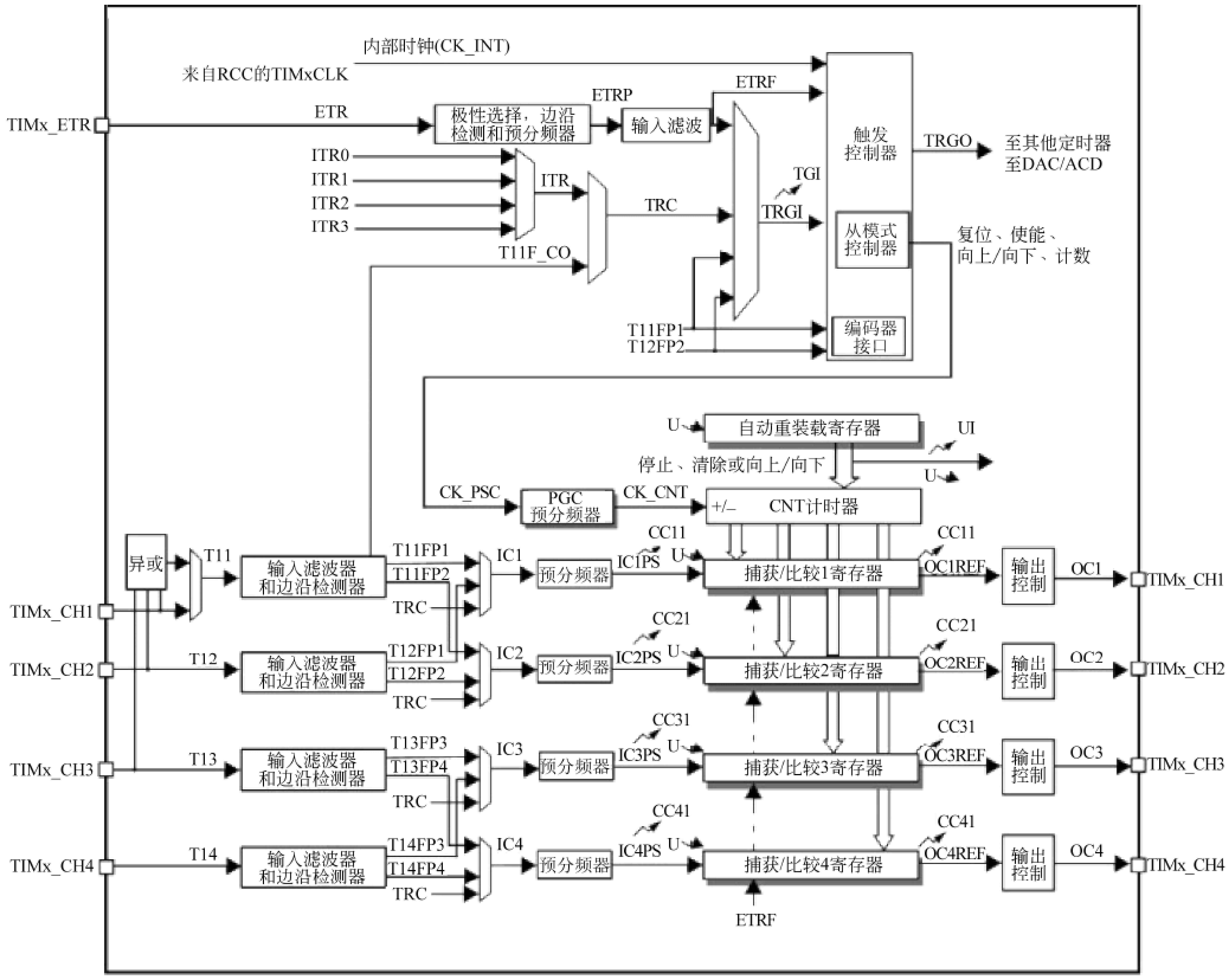
基本定时器只支持向上计数模式，计数器从 0 累加计数到自动重装载数值，产生一个计数器溢出事件，如果 TIMx_CR1 寄存器的 OPM(One Pluse Mode)被置为 1，则停止计数，CEN 被置 0；如果 OPM 被置为 0，则定时器重新从 0 开始计数。

7.5　通用定时器 TIM2～TIM4、TIM9～TIM11

通用定时器除了基本定时器的功能外，还包括向下、向上/向下自动装载计数，1～4 个独立通道用于输入捕获、输出比较、PWM 生成或单脉冲模式输出，可以使用外部信号控制定时器，支持定时器互连，多种中断/DMA 事件产生等功能。

STM32L152 有 6 个 16 位的通用定时器，分为两类，其中 TIM2～TIM4 连接在 APB1 总线上，各有 4 个独立的捕获比较通道，可以和 TIM9、TIM10 以及 TIM11 进行同步互联；TIM9～TIM11 连接在 APB2 总线上，TIM10 和 TIM11 只有 1 个捕获比较通道，TIM9 有 2 个捕获比较通道，可以被 TIM2、TIM3 和 TIM4 同步；同时 TIM9、TIM10 和 TIM11 这三个定时器可以使用 LSE 作为外部时钟源独立于总线时钟工作。

STM32L152 通用定时器的结构如图 7-13 所示，除了控制单元，时基单元外，还增加了时钟源选择、输入滤波和检测、捕获比较以及输出控制等单元。



注：Reg 根据控制位的设定，在U事件时传送预加载寄存器的内容至工作寄存器

事件

中断和DMA输出

图 7-13　通用定时内部结构

7.5.1　通用定时器时基单元

通用定时器的时基单元与基本定时器结构相同，由计数器寄存器 TIMx_CNT、预分频器寄存器 TIMx_PSC 和自动装载寄存器 TIMx_ARR 构成，但其计数模式支持三种模式：向上计数模式、向下计数模式和中央对齐模式。计数模式通过控制寄存器 TIMx_CR1 的计数方向域 DIR 和中央对齐模式域 CMS 进行设置（中央对齐模式下，不能写入 TIMx_CR1 中的 DIR 方向位，DIR 由硬件更新并指示当前的计数方向）。

每次计数器溢出时产生更新事件 UEV，当发生更新事件时，所有的影子寄存器都被立即更新，定时器状态寄存器 TIMx_SR 的更新中断标志位 UIF 被置为 1。

如图 7-13 所示，通用定时器的计数器由 CK_CNT 时钟驱动，当 TIMx_CR1 的 CEN 位置 1 时，计数器开始工作，CK_CNT 由预分频器给出，预分频器的时钟源 CK_PSC 可以选择多种时钟源提供。通用定时器中，TIM2、TIM3 和 TIM4 支持内部时钟（CK_INT）、外部输入脚触发（TI1 和 TI2）、内部定时器触发（ITR0～ITR3）和外部触发（ETR）四种。TIM9 支持上述四种，且其外部触发 ETR 在微控制器内部已被连接到 LSE 时钟；TIM10 和 TIM11 只支持 CK_INT、TI1 和 ETR，TIM10 和 TIM11 的 ETR 也连接到了 LSE 时钟；这样 TIM9、TIM10 和 TIM11 可以在微控制器休眠的状态下（CK_INT 被关闭）采用 LSE 时钟继续工作。

通常我们使用内部时钟 CK_INT 作为定时器的时钟源，如果从模式控制寄存器 TIMx_SMCR 的 SMS＝000，只要 CEN 位被写成 1，预分频器的时钟就由内部时钟 CK_INT 提供。

当 TIMx_SMCR 寄存器的 SMS＝111 时，外部触发模式 1 被选中，计数器可以在选定触发输入端的每个上升沿或下降沿计数。触发源共有 8 个，如图 7-14 所示，分别为 4 个其他定时器输出 ITRx、滤波后的定时器输入通道 TI1FP1、滤波后的定时器通道 TI2FP2、滤波后定时器边缘检测输入通道 TI1F_ED 和滤波后的外部触发源 ETRF，由 TIMx_SMCR 寄存器的 TS 域进行配置。如果选用 ITRx 作为时钟源，即选用了内部定时器触发作为时钟源，即定时器级联方式。在使用该模式前，需要先配置好外部触发源的信号。

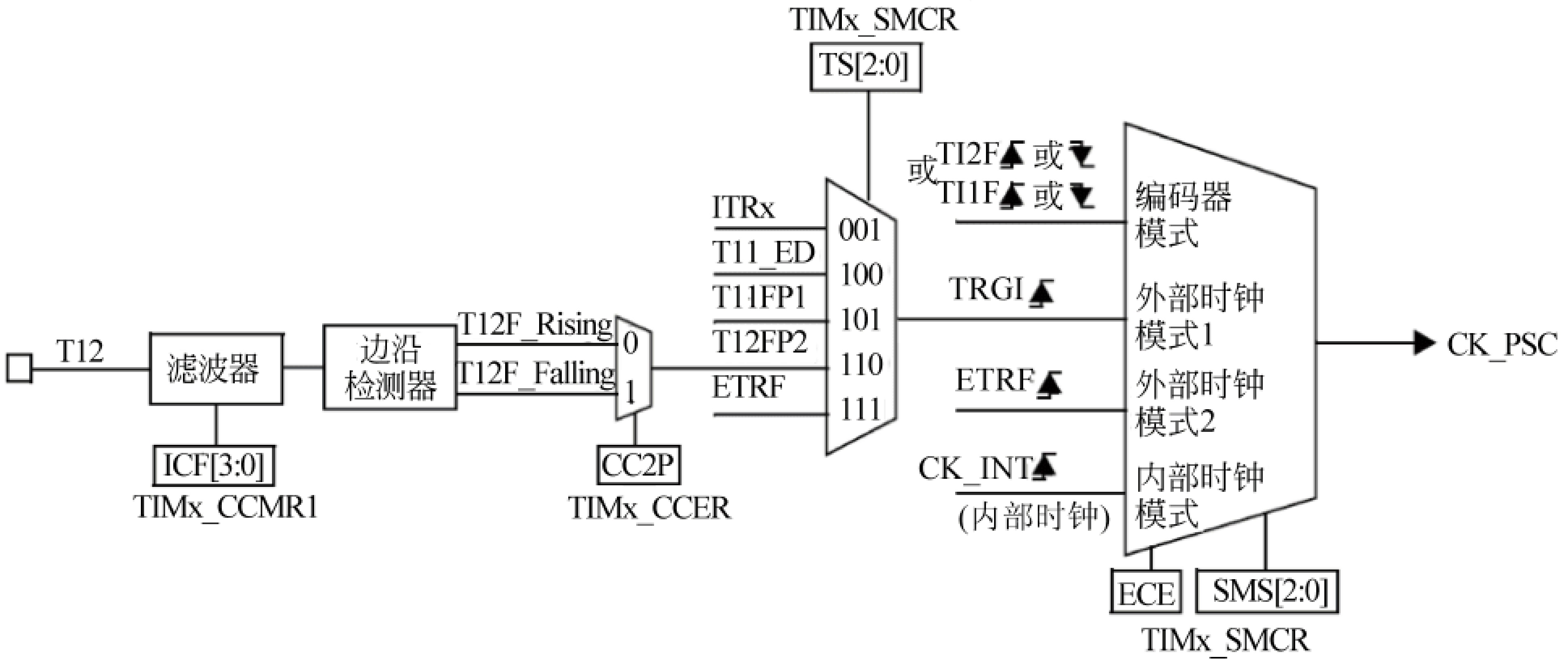


图 7-14　外部触发模式 1 的触发源

在外部触发模式 1 中，TIM10 和 TIM11 的外部时钟模式 1 触发源只能使用 TI1FP1、TI1F_ED 和 ETR。此外，不同于 TIM2、TIM3 和 TIM4 的时钟输入 TIx 只能由外部引脚输入，TIM9 的 TI1 输入时钟除了外部管输入脚 TIx 外（外部触发模式 1），还支持 LSE（外部触发模式 2）；TIM10 的 TI1 输入时钟除了外部引脚输入外，还支持 LSE、LSI 和 RTC 唤醒中断；TIM11 的 TI1 输入时钟除了外部引脚输入外，还支持 MSI 和 HSE_RTC；这些输入时钟由寄存器 TIMx_OR 进行配置。

如图 7-14 所示，要选择 TI2FP2 上升沿作为定时器的时钟源，则需要配置 TIMx_CCMR1 寄存器的的 CC2S、IC2F 以及 TIMx_CCER 寄存器的输入极性 CC2P、CC2NP 等，

然后在 TIMx_SMCR 中配置触发源为 TI1FP2(TS=110),模式为外部触发模式 1(SMS=111),配置 TIMx_CR1 寄存器 CEN=1 启动定时器后,TI2FP2 驱动定时器工作,其工作时序如图 7-15 所示(TIF 为触发中断标志,当检测到一个 TI2 输入时,TIF 被置 1,通过软件进行清 0)。

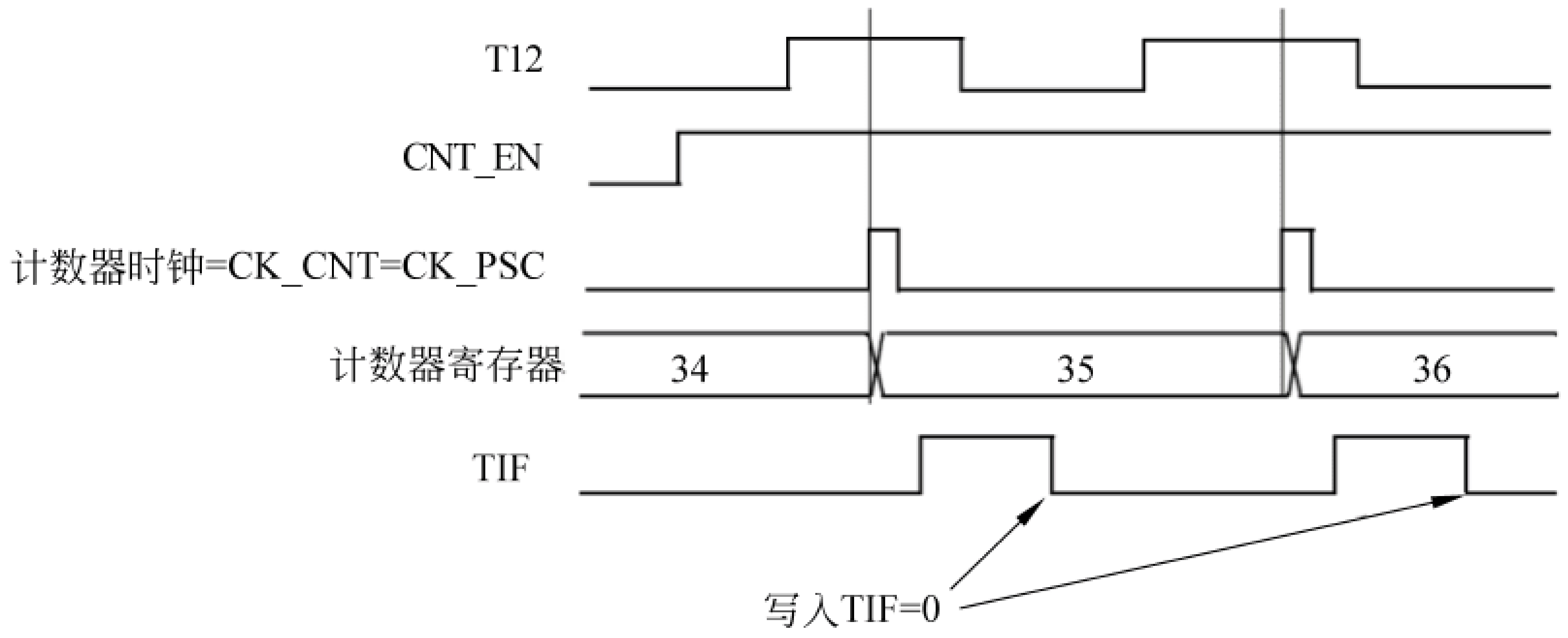


图 7-15 外部时钟模式下计数器时序

当 TIMx_SMCR 寄存器中的 ECE=1 时,外部触发模式 2 被选中,计数器在外部触发 ETR 的每一个上升沿或下降沿计数。如图 7-16 所示,外部时钟源 ETR 输入通路上带有分频器、滤波器,滤波器以一个可配置的基准频率对输入信号进行采样,当连续采样到 N 次有效电平时,认为一次有效的输入电平。

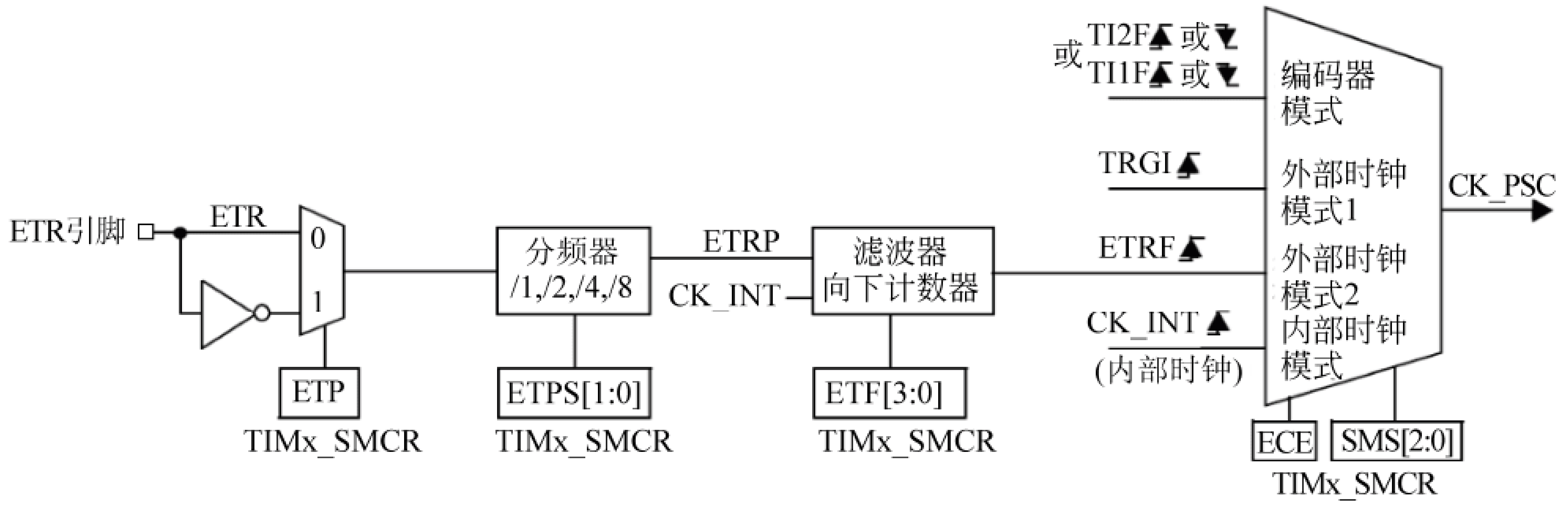


图 7-16 外部时钟源 ETR 触发电路

例如,配置外部 ETR 触发,2 个 ETR 上升沿进行一次计数,需要将 TIMx_SMCR 寄存器的 ETPS 置为 01,ETP 置为 0,配置 ECE=1 启用 ETR 外部时钟模式 2,启动定时器后,每两个 ETR 上升沿进行计数,时序如图 7-17 所示。

从图 7-14 可见,外部触发源 ETR 也可以在外部触发模式 1 下作为输入,此时和外部触发模式 2 下的功能一样。但如果需要使用 ETR 外部触发和从模式中的复位、触发、门控等进行组合应用,此时 SMS 已被配置为复位、触发或门控,无法配置 ETR 输入,此时可以配置 ECE 启用 ETR 作为时钟源。当外部模式 2 开启时,内部时钟和外部时钟模式 1 无效。

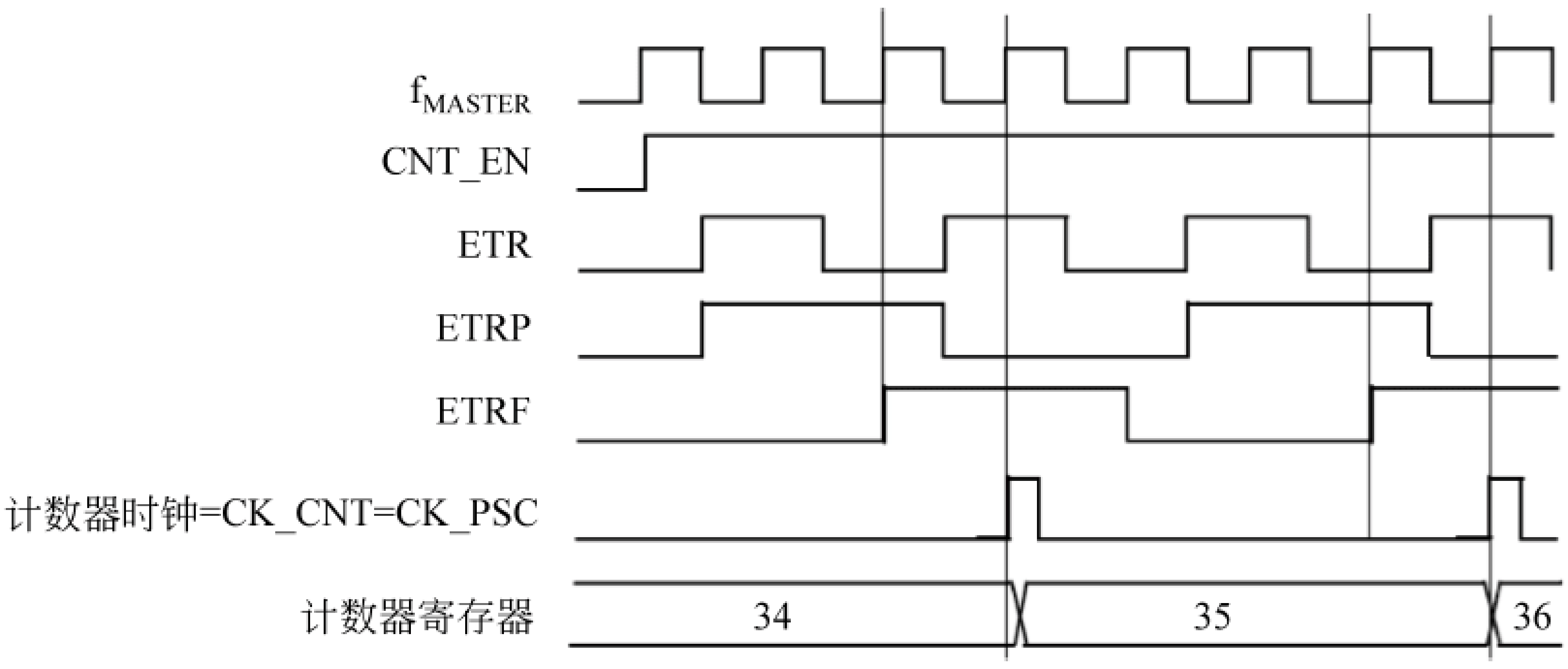


图 7-17 外部时钟触发模式 2 时序

7.5.2 通用定时器输入捕获和输出比较单元

输入捕获和输出比较是通用定时器较为复杂的功能，捕获可以用来对输入信号进行测量，输出比较可以用来产生特定的输出信号。每个定时器支持的捕获比较通道数量不同，但其组成结构类似。如图 7-9 所示，每个捕获通道的核心是一个捕获比较寄存器，它的输入部分主要是边沿检测电路，输出部分是输出控制电路。对于一个通道而言，只能选用输入捕获或输出比较中的一种模式。捕获/比较寄存器带有影子寄存器，读写过程仅操作其预装载寄存器。捕获模式下，捕获发生在影子寄存器上，然后再复制到预装载寄存器中；在比较模式下，预装载寄存器的内容被复制到影子寄存器中，影子寄存器和计数器进行比较判断。

1. 输入捕获

图 7-18 为捕获模式下输入电路的结构，输入部分对 TIx 引脚输入信号采样，产生一个滤波后的信号 TIxF，采样频率 fdts 由寄存器 TIMx_CR1 配置。然后，一个带极性选择的边缘检测器产生一个信号(TIxFPx)。通道选择器决定送到比较通道的信号源，该信号通过预分频后作为最终的输入信号 ICxPS 送到捕获/比较寄存器。滤波的目的是通过多次采样减少信号高频部分带来的抖动干扰。

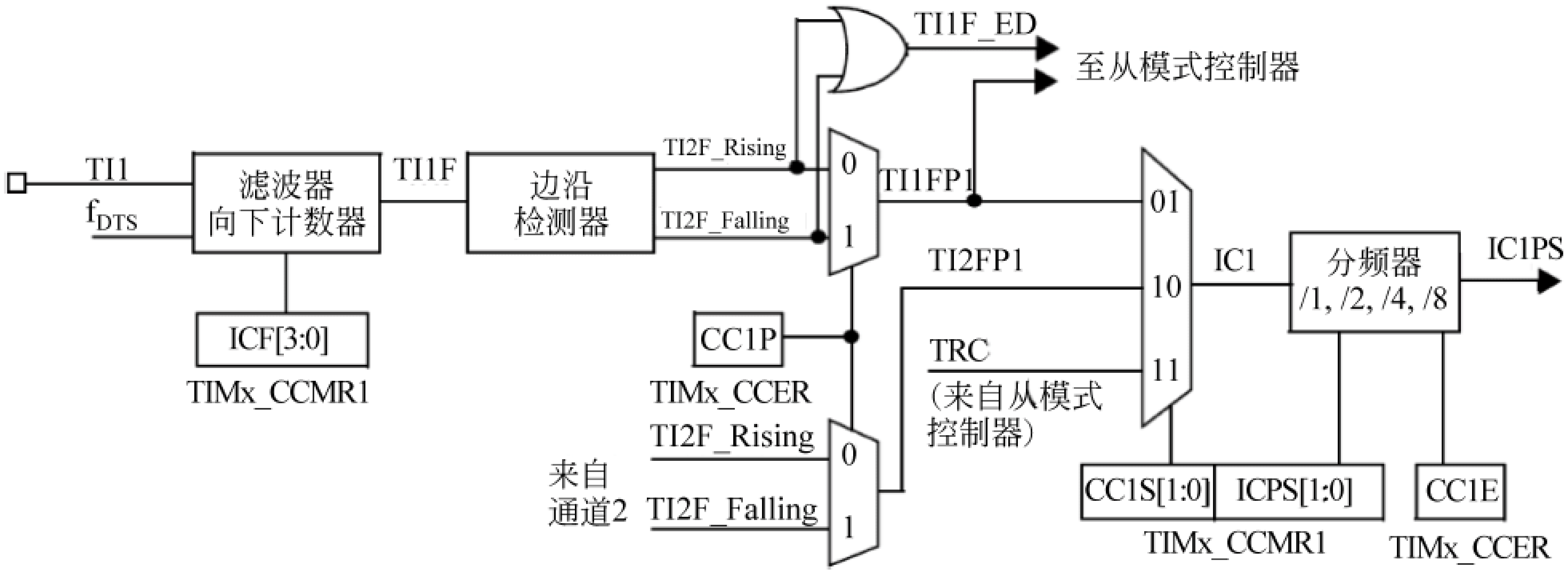


图 7-18 输入捕获电路结构

在输入捕获模式下，当检测到ICx信号上相应的边沿后，计数器的当前值被锁存到捕获/比较寄存器(TIMx_CCRx)中，同时产生捕获事件，置TIMx_SR寄存器相应的标志CCxIF为1，如果使能了中断或DMA，则将产生中断或者DMA操作。如果捕获事件发生时CCxIF标志已经为1，则重复捕获标志CCxOF被置1。读取存储在TIMx_CCRx寄存器中数据时CCxIF被清0。

配置示例：在TI1输入端口输入一个频率为1MHz的方波，输入信号在最多5个内部时钟周期的时间内抖动，配置输入捕获寄存器在信号的上升沿捕获计数器的值并计算周期。

(1) 配置TI1输入端口的GPIO相关寄存器，将端口设置为复用输入模式。

(2) 选择输入端口：配置TIMx_CCR1寄存器的CC1S=01，选择TI1作为捕获输入源。

(3) 配置输入滤波：配置滤波采样频率为CK_INT，输入信号抖动在5个时钟周期内，因此TIMx_CCMR1寄存器中的IC1F置为0011，即连续采样8次对输入信号上升沿进行判断。

(4) 选择转换边沿：在TIMx_CCER寄存器中写入CC1P=0，捕获上升沿。

(5) 配置预分频器：如果希望n(1,2,4,8)次信号上升沿捕获产生一个有效的电平转换时刻，配置预分频器IC1PS=00,01,10和11。

(6) 开启捕获：设置TIMx_CCER寄存器的CC1E=1，允许捕获。

(7) 开启中断：设置TIMx_DIER寄存器中的CC1IE和CC1DE位允许相关中断请求和DMA请求。

(8) 记录两次捕获的计数值，即可算出输入信号的频率。

2. 输出比较

如图7-19所示，输出比较电路的输出结果实际是由OCxREF决定的，比较寄存器和计数器的值相等时，输出一个OCxREF电平，电平由OCxM决定，可以是保持原有电平，设置为低电平，设置为高电平或者进行翻转，这个输出电平也可以被外部触发时钟源进行控制。OCxREF的信号可以被极性选择器进行翻转，即低电平变高电平，高电平变低电平，而电平信号是否输出最终取决于输出使能电路，只有TIMx_CCER的CCxE域为1时OC1才真正输出。

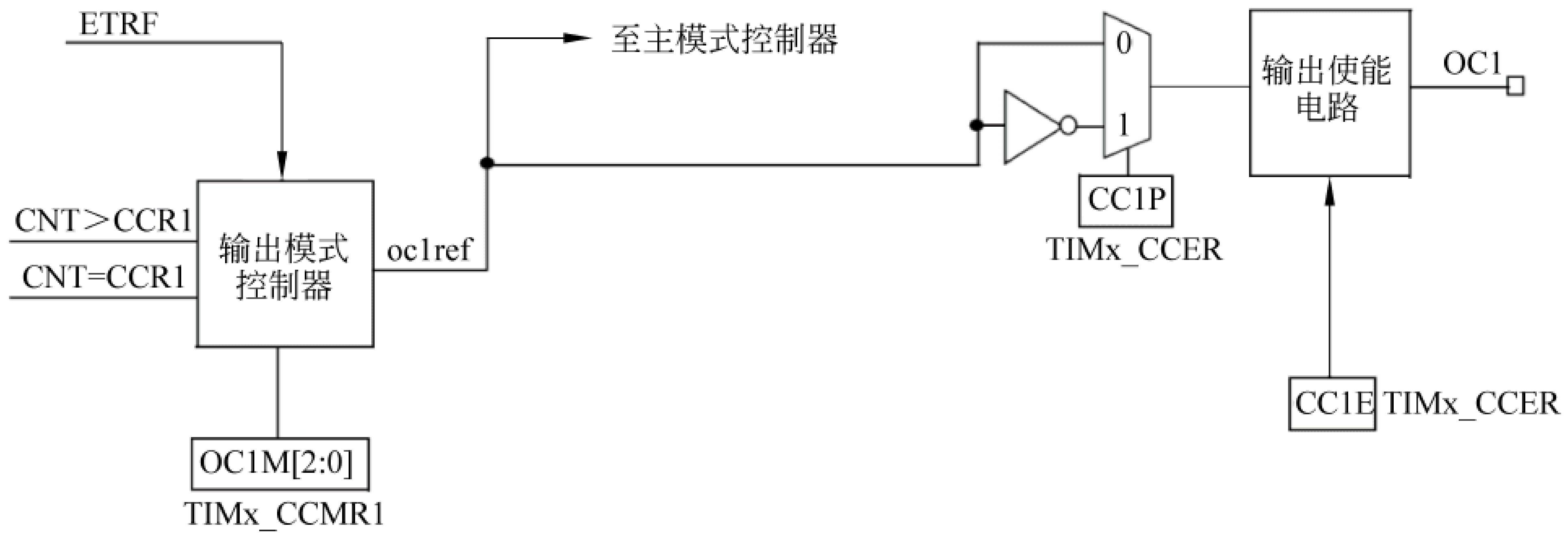


图7-19　输出比较电路

当计数器与捕获/比较寄存器的内容相同时，输出比较功能做如下操作：

- 根据比较模式(TIMx_CCMRx寄存器中的OCxM域)和输出极性(TIMx_CCER寄

存器中的 CCxP 位)的定义,将 OCxREF 的值输出到对应的引脚上。

• 设置中断状态寄存器中的标志位(TIMx_SR 寄存器中的 CCxIF 位)。
• 若设置了中断屏蔽(TIMx_DIER 寄存器中的 CCxIE 位)允许中断,则产生一个中断请求。

在输出比较模式下,只有比较事件才会影响输出结果,更新事件对 OCxREF 和 OCx 输出没有影响。

输出比较模式的配置步骤一般为:

(1) 选择计数器时钟(内部、外部,设置预分频器);

(2) 将相应的数据写入 TIMx_ARR 和 TIMx_CCRx 寄存器中;

(3) 设置 CCxIE、CCXDE 是否产生中断请求/或 DMA 请求;

(4) 配置输出模式 OCxM;

(5) 设置 TIMx_CR1 寄存器的 CEN 位启动计数器。

3. PWM 输出

PWM 指脉冲宽度调制,也就是占空比可变的脉冲波形,PWM 被广泛应用在电机控制、频率调节等领域。PWM 模式下,可以产生一个由 TIMx_ARR 寄存器确定频率、由 TIMx_CCRx 寄存器确定占空比的信号。例如,当 CNT 的值小于 TIMx_CCRx 中的值时候输出高电平或低电平,当大于的时候反向,如图 7-20 所示。

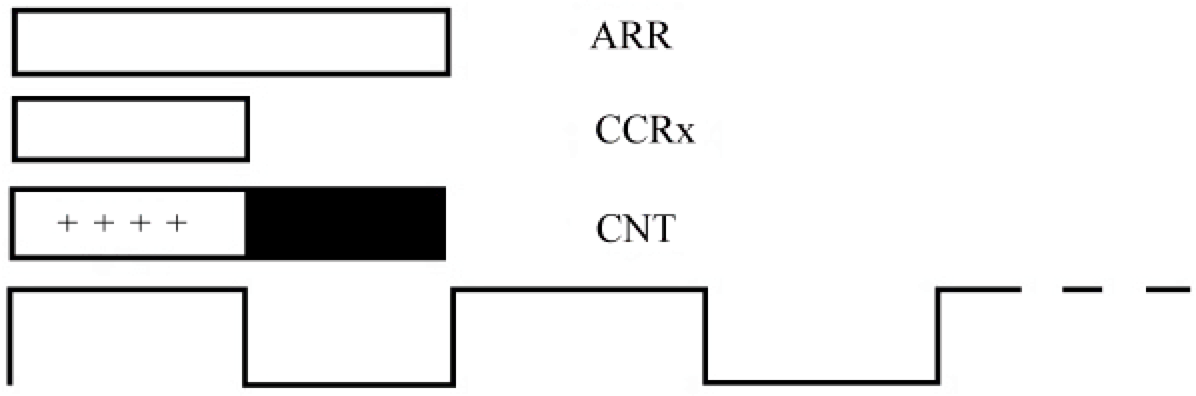


图 7-20　PWM 示意图

TIMx_ARR 寄存器设定脉冲周期,TIMx_CCR 设定占空比,CNT 计数器最大值为 TIMx_ARR,由于 TIM 有多于 1 个的捕获比较通道,因此一个定时器可以同时输出多个同一频率但占空比不同的 PWM 波形。

TIMx_CCMRx 寄存器中的 OCxM 支持两种 PWM 模式:

(1) PWM 模式 1:在向上计数时,一旦 CNT<CCRx,通道 x 为高电平(OC1REF=1),否则为低电平(OC1REF=0);在向下计数时,一旦 CNT>CCRx,通道 x 为低电平,否则为高电平(OC1REF=1)。

(2) PWM 模式 2:在向上计数时,一旦 CNT<CCRx,通道 x 为低电平,否则为高电平;在向下计数时,一旦 CNT>CCRx,通道 x 为高电平,否则为低电平。

在 PWM 模式下,必须设置 TIMx_CCMRx 寄存器 OCxPE 位为使能预装载寄存器,设置 TIMx_CR1 寄存器的 ARPE 位使能自动重装载的预装载寄存器,因此启用计数器之前须通过设置 TIMx_EGR 寄存器中的 UG 位来手动初始化所有的寄存器。

图 7-21 为 ARR 配置为 8 时,PWM 模式 1 的示例。当 CNT<CCRx 时,PWM 输出

OCxREF 为高，否则为低。如果 CCRx 的值大于自动重装载值(TIMx_ARR)，则输出一个占空比 100%的信号，即 OCxREF 始终为 1。如果 CCRx=0，则产生占空比 0%的信号，即 OCxREF 始终保持为 0。

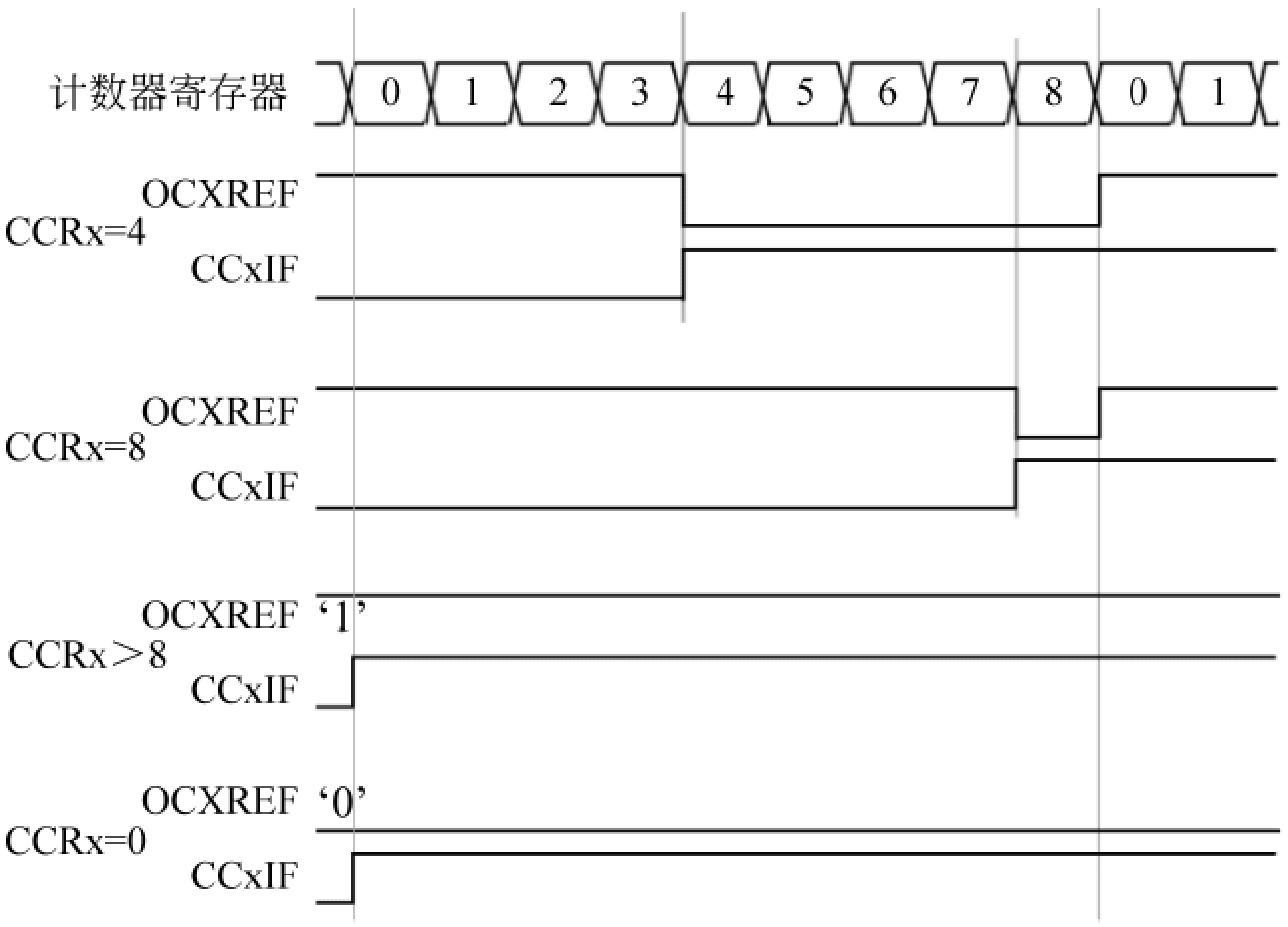


图 7-21　PWM 模式 1 时序

4. PWM 输入模式

PWM 输入模式用来测量 PWM 的周期和占空比，该模式是输入捕获模式的一个特例，其与不同输入捕获的区别在于：

- 一个 TIx 的输入被连接到两个不同的 ICx 通道上；
- 两个 ICx 输入信号配置为边沿有效，但极性相反；
- 一个 TIxFP 信号被作为触发输入信号，从模式控制器被配置成复位模式。

由于只有 TI1FP1 和 TI2FP2 连到了从模式控制器，所以 PWM 输入模式只能使用 TIMx_CH1 /TIMx_CH2 信号。将输入 PWM 信号连接到 TI1，则信号周期存储在 TIMx_CCR1 寄存器，占空比存储在 TIMx_CCR2 寄存器，具体配置步骤如下：

(1) 选择 TIMx_CCR1 的有效输入，TIMx_CCMR1 的 CC1S=01(选择 TI1)；

(2) 选择 TI1FP1 的极性为上升沿有效，置 CC1P=0；

(3) 选择 TIMx_CCR2 的有效输入，TIMx_CCMR1 的 CC2S=10(选择 TI1)；

(4) 选择 TI1FP2 的有效极性为下降沿有效，置 CC2P=1；

(5) 选择外部触发时钟为 TIFP1，置 TIMx_SMCR 的 TS=101；

(6) 配置从模式控制器为复位模式，置 TIMx_SMCR 中的 SMS=100；

(7) 使能捕获，置 TIMx_CCER 的 CC1E=1 且 CC2E=1。

如图 7-22 所示，当 T1 检测到上升沿时，复位定时器，CNT 从 0 开始计数，当下降沿到达时，IC2 捕获到此时的计数值，即为 PWM 高电平宽度，当下一个上升沿到达时，IC1 捕获到此时的计数值，即为 PWM 的周期，通过定时器时钟频率和捕获的计数值即可算出 PWM 信号的实际周期和占空比。

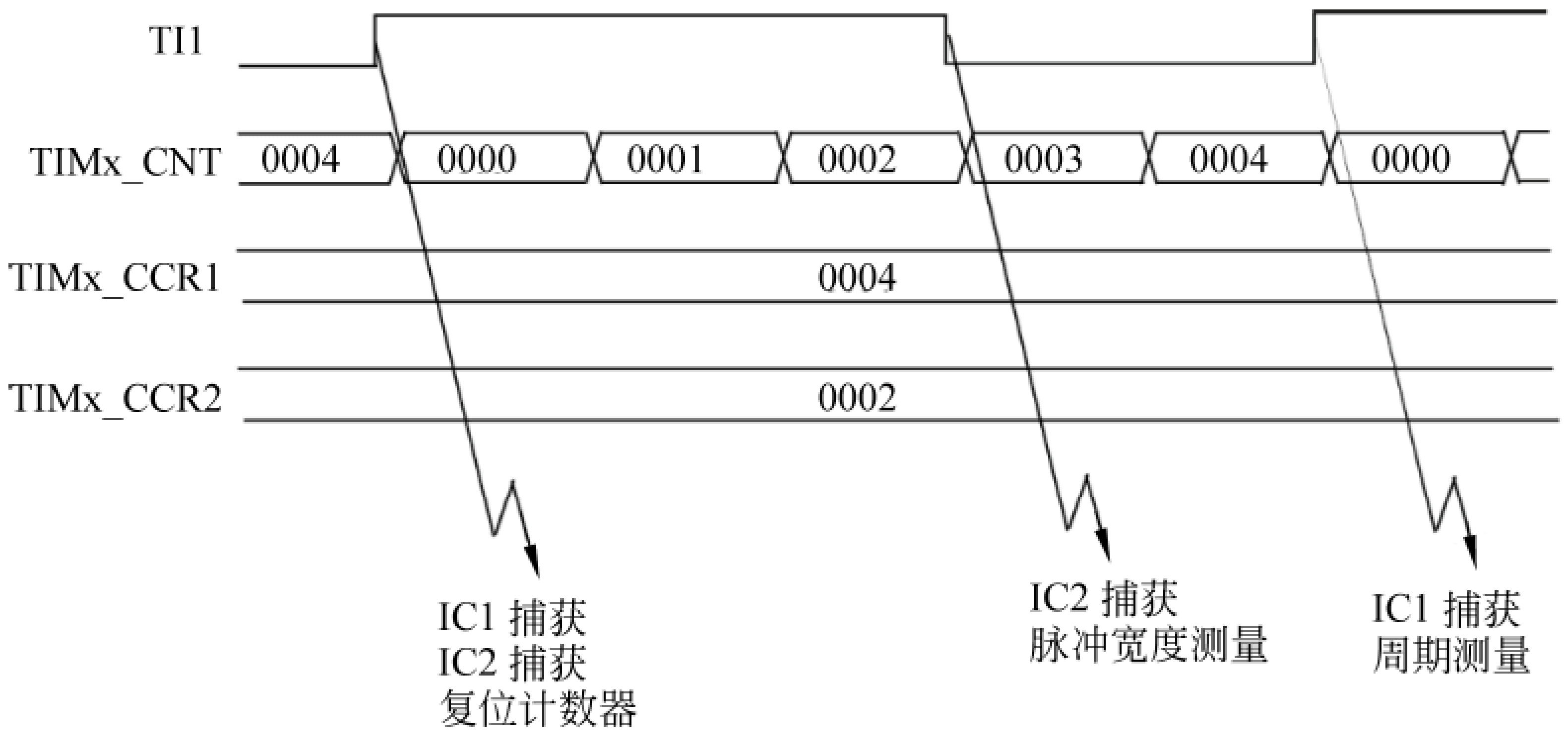


图 7-22　PWM 输入测量

7.5.3　TIMx 的外部触发同步模式

TIMx 在从模式可以通过外部触发信号同步，从模式包括复位、门控和触发模式。

1. 复位模式

外部触发输入事件发生时，计数器和它的预分频器被重新被初始化，如果 IMx_CR1 寄存器的 URS 位为 0，即从模式控制器可以产生更新事件，此时还产生一个更新事件 UEV；然后所有的预装载寄存器(TIMx_ARR，TIMx_CCRx)都被更新。

如图 7-23 所示，TI1 上升沿作为触发，产生 UEV 事件，计数器被清 0，TIF 标志表示产生了一个触发中断，如果中断允许，则向 MCU 发起中断请求。

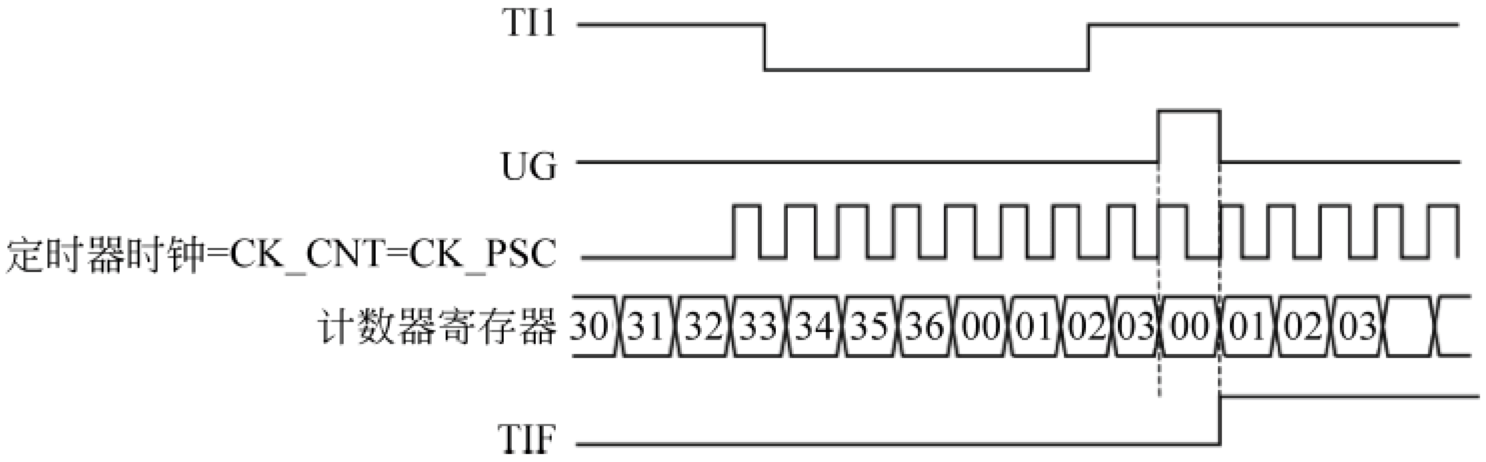


图 7-23　复位模式的控制时序

2. 门控模式

计数器的使能依赖于选中的输入端的电平。如下图 7-24 所示，TI1 的低电平作为触发源，配置通道 1 作为 TI1 低电平检测，当 TI1 为高电平时，计数器停止计数，当检测到 TI1 变为低电平后，计数器立即开始工作。在计数器停止和启动均会引起 TIF 标志置 1，表示产生一个触发中断。

3. 触发模式

计数器的使能依赖于选中的输入端上的事件。如图 7-25 所示，配置 TI2 的上升沿作为外部触发源，当检测到上升沿时，计数器启动，开始计数，并设置触发中断标志 TIF，否则定

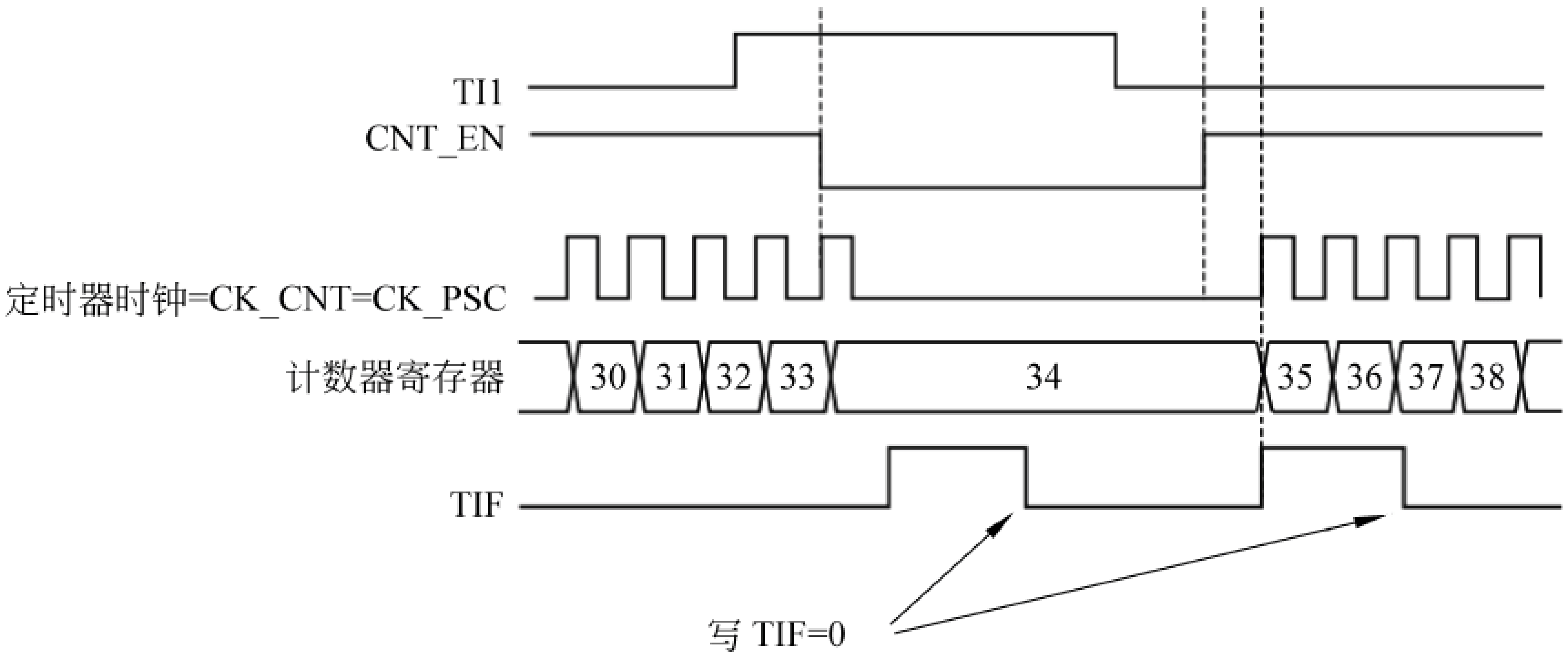


图 7-24　门控模式的控制时序

时器停止工作。

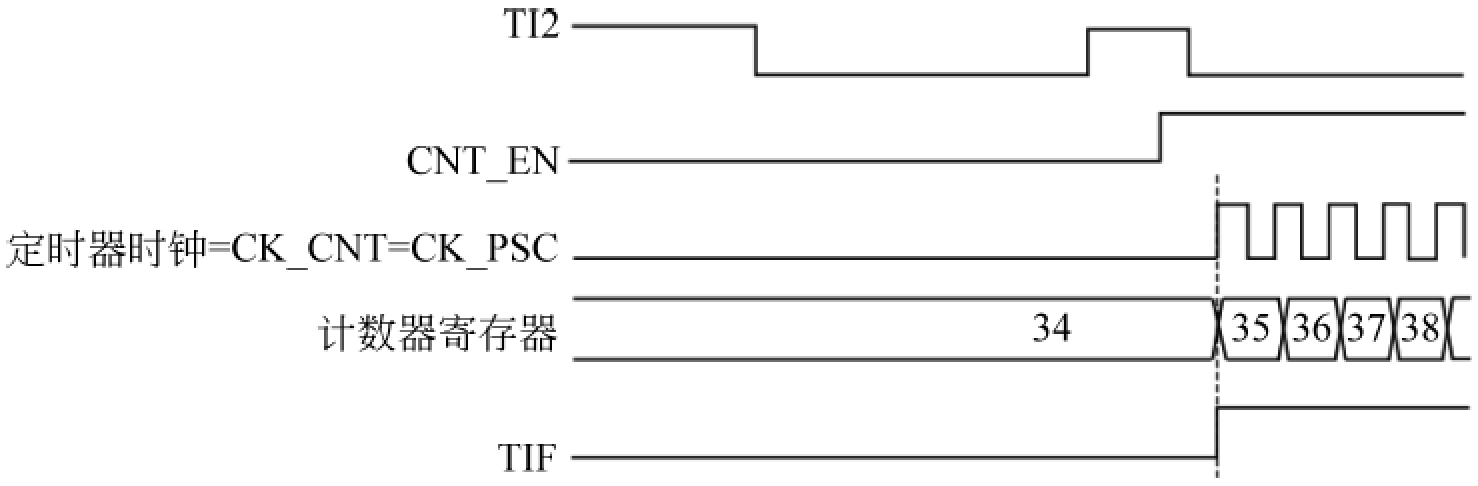


图 7-25　触发模式的控制时序

4. 外部时钟 2+触发模式

外部时钟模式 2 可以与另一种从模式(外部时钟模式 1 和编码器模式除外)一起使用。这时,ETR 信号被用作外部时钟的输入,在复位模式、门控模式或触发模式可以选择另一个输入作为触发输入。如图 7-26 所示,TI1 上升沿作为触发信号,ETR 作为定时器时钟,当检测到 TI1 的上升沿后,定时器启动,在 ETR 的控制下开始计数。

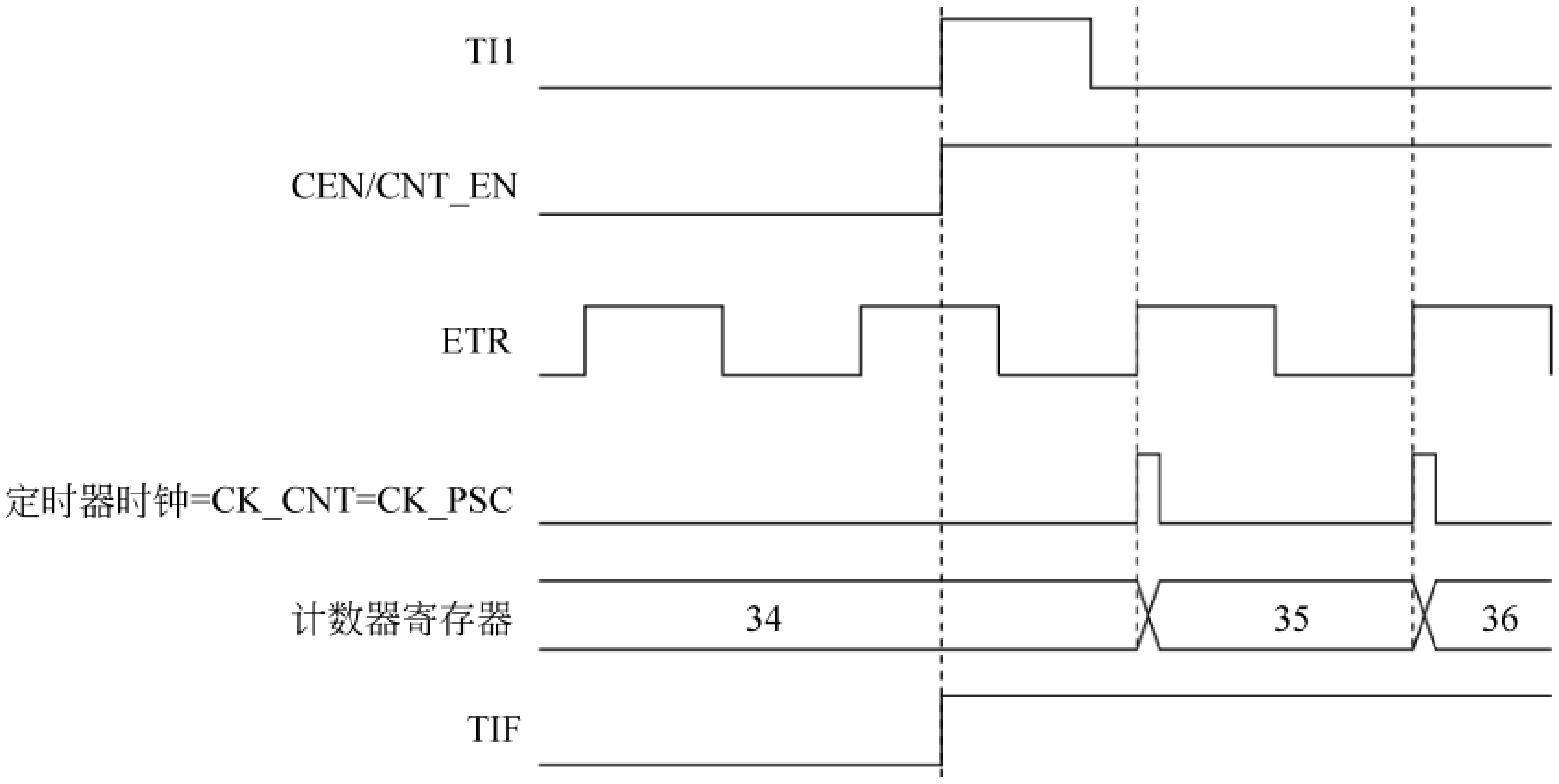


图 7-26　外部时钟模式 2+触发模式的控制时序

7.6　定时器寄存器

通用定时器包括 16 个寄存器，TIM2～TIM4，TIM6～TIM7 以及 TIM9～TIM11 各自对每个寄存器的域的支持有所不同，TIM9～TIM11 还有自己特殊的寄存器，下面对常用寄存器进行介绍，其中不同定时器对不同域的支持在寄存器域说明中进行标注。定时器常用寄存器如表 7-4 所示。

表 7-4　定时器寄存器及其功能

寄存器名	读写权限	功　能
TIMx_CR1	RW	定时器控制寄存器 1
TIMx_CR2	RW	定时器控制寄存器 2
TIMx_SMRC	RW	定时器从模式选择寄存器
TIMx_DIER	RW	定时器 DMA 和中断使能寄存器
TIMx_SR	RW	定时器状态寄存器
TIMx_EGR	W	定时器事件产生寄存器
TIMx_CCMR1	RW	捕获/输出模式寄存器 1
TIMx_CCMR2	RW	捕获/输出模式寄存器 2
TIMx_CCER	RW	捕获/输出使能寄存器
TIMx_CNT	RW	计数器
TIMx_PSC	RW	预分频寄存器
TIMx_ARR	RW	自动重装载寄存器
TIMx_CCR1	RW	通道 1 捕获/比较寄存器
TIMx_CCR2	RW	通道 2 捕获/比较寄存器
TIMx_CCR3	RW	通道 3 捕获/比较寄存器
TIMx_CCR4	RW	通道 4 捕获/比较寄存器

1. 控制寄存器 TIMx_CR1

控制寄存器 TIMx_CR1 是 16 位寄存器，如图 7-27 所示，其有效域包括：

CKD[1：0]：时钟分频因子，用于配置 ETR、TIx 的数字滤波器采样频率 fdts 和定时器

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CKD[1:0]		ARPE	CMS		DIR	OPM	URS	UDIS	CEN
						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 7-27　控制寄存器 1

内部时钟(CK_INT)频率之间的分频比例。00 表示不分频,01 表示二分频,10 表示 4 分频。

ARPE:自动重装载预装载允许位,0 表示 TIMx_ARR 寄存器没有缓冲,写入 ARR 寄存器将直接改变影子寄存器的值,1 表示 TIMx_ARR 寄存器被装入缓冲器,只有事件发生时 ARR 寄存器值才会被加载到影子寄存器。

CMS[1:0]:选择中央对齐的四种模式,分别为不使用中央对齐,中央对齐模式 1、中央对齐模式 2 和中央对齐模式 3,具体见 STM32L1xx 参考手册文档。

DIR:计数方向,0 表示向上计数,1 表示向下计数,当选用中央对齐模式时,该位为只读,表示计数方向。当配置向上计数时,配置 CMS=00,DIR=0。

OPM:单脉冲模式,0 表示在发生更新事件时,计数器不停止,1 表示在下一次更新事件发生时清除 CEN 位,计数器停止。

URS:更新请求源,设置产生中断的事件源,0 表示计数器溢出、设置 UG 位以及从模式控制器产生的更新都可以产生中断,1 表示只有计数器溢出才产生更新中断。

UDIS:禁止更新,0 表示允许产生更新事件,1 表示不产生更新事件。

CEN:计数器启动位,0 表示禁止计数器,1 表示启动计数器。

其中,基本定时器 TIM6 和 TIM7 没有 CMS 和 DIR 域。

2. 控制寄存器 TIMx_CR2

控制寄存器 TIMx_CR2 是 16 位寄存器,如图 7-28 所示,其有效域包括:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								TI1S	MMS[2:0]			CCDS	Reserved		
								rw	rw	rw	rw	rw			

图 7-28 控制寄存器 2

TI1S:TI1 输入源选择,0 表示 TIMx_CH1 引脚连到 TI1 输入,1 表示 TIMx_CH1、TIMx_CH2 和 TIMx_CH3 引脚经异或后连到 TI1 输入。

MMS[2:0]:主模式选择,用于选择在主模式下送到从定时器的同步信息 TRGO 的来源。

- 000 表示复位,TIMx_EGR 寄存器的 UG 位被用于作为触发输出(TRGO)。
- 001 表示使能,计数器使能信号 CNT_EN 被用于作为触发输出(TRGO)。
- 010 表示更新,更新事件 UE 被选为触发输入(TRGO)。
- 011 表示比较脉冲,在发生一次捕获或一次比较成功时,当要设置 CC1IF 标志时,触发输出送出一个正脉冲(TRGO)。
- 100~111 表示 OC1REF~OC4REF 信号被用于作为触发输出(TRGO)。

其中,基本定时器 TIM6 和 TIM7 只有 MMS 域。TIM10、TIM11 没有此寄存器。

3. 从模式控制寄存器(TIMx_SMCR)

从模式控制寄存器 TIMx_SMCR 为 16 位寄存器,如图 7-29 所示,其主要域包括:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ETP	ECE	ETPS[1:0]		ETF[3:0]				MSM	TS[2:0]			OCCS	SMS[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 7-29 从模式控制寄存器

ETP：外部触发极性，用于选择是用 ETR 还是 ETR 的反相来作为触发操作，0 表示 ETR 不反相，高电平或上升沿有效，1 表示 1，ETR 反相，低电平或下降沿有效。

ECE：外部时钟使能位，1 表示启用外部时钟模式 2，计数器由 ETRF 信号上的任意有效边沿驱动，0 表示禁用外部时钟模式 2。

ETPS[1：0]：外部触发预分频系数，外部触发信号 ETRP 的频率最高不能超过 CK_INT/4。ETRP 过快时，使用预分频降低 ETRP 的频率。00 表示关闭预分频，01～11 分别表示 2、4、8 分频。

ETF[3：0]：外部触发滤波，定义对 ETRP 信号采样的频率和对 ETRP 数字滤波的带宽。数字滤波带宽用 N 表示，即 N 个事件后会产生一个输出的跳变，N 的取值为 2、4、5、6、8。采样频率可以选择 fCK_INT 或 fDTS 的 2、4、8、16 分频，其中 fDTS 由 TIMx_CR1 的 CDK 配置。具体见 STM32L1xx 参考手册文档。

TS[2：0]：外部模式 1 的触发时钟源选择，000-011 表示内部定时器 TIM1～TIM4 作为时钟源，100～110 为输入通道 1 的不同信号作为时钟源，111 表示外部触发输入 ETRF。TS 和 SMS 配合使用，SMS＝000 时 TS 无效。

SMS[2：0]：从模式选择，000 表示不使用从模式，预分频器直接由内部时钟驱动，100～110 为复位模式、门控模式和触发模式，111 表示使用外部时钟模式 1，具体外部时钟源由 TS 选择。

基本定时器 TIM6、TIM7 没有这个寄存器，通用定时器 TIM9 没有 OCCS 域。TIM10、TIM11 只有 8～15 位，没有 MSM、TS、OCCS 以及 SMS 域。

4．DMA/中断使能寄存器 TIMx_DIER

中断使能寄存器为 16 位寄存器，如图 7-30 所示，其有效域为：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	TDE	Res	CC4DE	CC3DE	CC2DE	CC1DE	UDE	Res.	TIE	Res	CC4IE	CC3IE	CC2IE	CC1IE	UIE
	rw		rw	rw	rw	rw	rw		rw		rw	rw	rw	rw	rw

图 7-30　中断使能寄存器

TDE：允许触发 DMA 请求，0 表示禁止，1 表示允许。

CC4DE～CC1DE：允许捕获/比较通道 x 的 DMA 请求，0 表示禁止，1 表示允许。

UDE：允许更新的 DMA 请求，0 表示禁止，1 表示允许。

TIE：触发中断使能，0 表示禁止，1 表示允许。

CC4IE～CC1IE：允许捕获/比较通道 x 中断，0 表示禁止，1 表示允许。

UIE：允许更新中断，0 表示禁止，1 表示允许。

其中，基本定时器 TIM6 和 TIM7 只有 UDE 和 UIE 两个域。TIM9 只有 TIE、CC2IE、CC1IE 和 UIE 域，TIM10 和 TIM11 只有 CC1IE 和 UIE 域。

5．状态寄存器 TIMx_SR

状态寄存器 TIMx_SR 为 16 位寄存器，如图 7-31 所示，其有效域包括：

CC4OF～CC1OF：捕获/比较通道重复捕获标记，0 表示无重复捕获，1 表示重复捕获，

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CC4OF	CC3OF	CC2OF	CC1OF	Reserved		TIF	Res	CC4IF	CC3IF	CC2IF	CC1IF	UIF
			rc_w0	rc_w0	rc_w0	rc_w0			rc_w0		rc_w0	rc_w0	rc_w0	rc_w0	rc_w0

图 7-31　状态寄存器

在输入捕获下，若计数器的值被捕获到 TIMx_CCR1 寄存器，但 CC1xF 的状态已经为 1，则 CCxOF 标记由硬件置 1，写 0 可清除该位。

TIF：触发器中断标记，当发生触发事件时由硬件置 1，软件写 0 清除。

CC4IF～CC1IF：捕获/比较通道中断标记，当通道 CCx 配置为输出模式，计数器值与比较值匹配时该位由硬件置 1，软件写 0 清除；当通道 CCx 配置为输入模式，当捕获事件发生时该位由硬件置 1，可由软件清 0 或通过读 TIMx_CCRx 清 0。

UIF：更新中断标记，当产生更新事件时该位由硬件置 1，它由软件清 0。

其中，基本定时器 TIM6、TIM7 只有 UIF 域，TIM9 只有 CC2OF、CC1OF、TIF、CC2IF、CC1IF 以及 UIF 域，TIM10 和 TIM11 只有 CC1OF、CC1IF 和 UIF 域。

6. 事件产生寄存器（TIMx_EGR）

事件产生寄存器用于软件触发事件，如图 7-32 所示，其有效域包括：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved									TG	Res.	CC4G	CC3G	CC2G	CC1G	UG
									w		w	w	w	w	w

图 7-32　事件产生寄存器

TG：产生触发事件，该位写 1 产生一个触发事件，由硬件自动清 0。

CC4G～CC1G：产生捕获/比较通道事件，该位写 1 产生一个捕获/比较事件，由硬件自动清 0。

UG：产生更新事件，该位写 1 产生更新事件，重新初始化计数器，预分频器计数器也被清 0 但预分频系数不变。

其中，基本定时器 TIM6 和 TIM7 只有 UG 域，TIM10 和 TIM11 只有 CC1G 和 UG 域，TIM9 只有 TG、CC2G、CC1G 和 UG 域。

7. 捕获/比较模式寄存器 TIMx_CCMR1

捕获比较寄存器 TIMx_CCMR1 用于配置捕获/比较通道 1 和 2，在配置成输入捕获和比较输出时使用同一个寄存器，但其域有不同的定义，其中 CCxS 域在两种模式下相同，CCxS 用于定义通道的方向。如图 7-33 所示，配置成输入捕获时，使用第二行的域定义，比较输出时，采用第一行的域定义。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OC2CE	OC2M[2:0]			OC2PE	OC2FE	CC2S[1:0]		OC1CE	OC1M[2:0]			OC1PE	OC1FE	CC1S[1:0]	
IC2F[3:0]				IC2PSC[1:0]				IC1F[3:0]				IC1PSC[1:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 7-33　捕获/比较模式寄存器

TIM6 和 TIM7 没有此寄存器，TIM10 和 TIM11 没有高 8 位。

CC1S[1：0]：捕获/比较 1 选择，定义通道的方向（输入输出）以及输入脚的选择，配置为 00 时通道为输出，01 表示输入，IC1＝TI1，10 表示输入，IC1＝TI2，11 时通道为输入，输入源有 TIMx_SMCR 的 TS 确定。CCxS 仅在通道关闭时（TIMx_CCER 寄存器的 CC1E＝0）才可以配置。

CC2S[1：0]：捕获/比较 2 选择，配置为 01 表示输入，IC2＝TI2，10 表示输入，IC2＝TI1，其余与 CC1S 相同。

输出比较模式下各个域的定义如下：

OCxCE：输出比较通道 x 清 0 使能，1 表示一旦检测到 ETRF 输入高电平，OCxREF 置为 0。

OCxM[2：0]：输出比较通道 x 的 OCxREF 模式选择，OCxREF 决定了 OCx 的输出值，而 OCx 的实际输出电平取决于 TIMX_CCER 寄存器的 CCxP 极性设置是否对 OC1REF 进行翻转。该域配置为 000 表示冻结，计数器与比较寄存器匹配不影响 OCxREF；001 表示计数器与比较寄存器匹配时，OC1REF 置为为高电平；010 表示匹配时置为低电平；011 表示匹配时翻转 OCxREF 的电平；100 和 101 分别表示强制输出为低电平和高电平；110 和 111 表示 PWM 的模式 1 和模式 2。

OCxPE：输出比较通道 x 预装载使能，0 表示禁止 TIMx_CCRx 寄存器的预装载功能，写入 TIMx_CCRx 寄存器的数值立即生效；1 表示开启 TIMx_CCRx 寄存器的预装载功能，TIMx_CCRx 的预装载值在更新事件到来时被传送至当前寄存器中。

OCxFE：输出比较通道 x 快速使能，仅在 PWM1 和 PWM2 模式使用。

输入捕获模式下寄存器的域定义如下：

ICxF[3：0]：输入捕获通道 x 的滤波器设置，定义了 TI1 输入的采样频率及数字滤波器长度。数字滤波器由一个事件计数器组成，记录到 N 个事件后会产生一个输出的跳变，N 的取值为 2，4，5，6，8，采样频率可以选择 fCK_INT 或 fDTS 的 2、4、8、16 分频，其中 fDTS 由 TIMx_CR1 的 CDK 配置，具体参考 STM32L1xx 参考手册。

ICxPSC[1：0]：输入/捕获通道 x 的预分频器，当 TIMx_CCER 寄存器的 CCxE 被清 0，预分频器复位。00 表示无预分频 01～11 分别表示 2、4、8 个事件触发一次捕获。

8. 捕获/比较模式寄存器 TIMx_CCMR2

TIMx_CCMR2 与 TIMx_CCMR1 的域定义类似，区别在于 CC3S 配置为 01 时表示输入 TI3 连接到 IC3，10 表示 TI4 连接到 IC3；而 CC4S 配置为 01 时表示 TI4 被连接到 IC4，10 表示 TI3 被连接到 IC4。

9. 捕获/比较使能寄存器 TIMx_CCER

如图图 7-34 所示，TIMx_CCER 的有效域定义如下：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CC4NP	Res.	CC4P	CC4E	CC3NP	Res.	CC3P	CC3E	CC2NP	Res.	CC2P	CC2E	CC1NP	Res.	CC1P	CC1E
rw		rw	rw	rw		rw	rw	rw		rw	rw	rw		rw	rw

图 7-34　捕获/比较使能寄存器

CC4P～CC1P：捕获/比较通道 x 的输出极性，通道 x 配置为输出时，该位配置为 0 表示 OCx 高电平有效，1 表示低电平有效；通道 x 配置为输入时，该位表示是选择 ICx 还是 ICx 的反相信号作为触发或捕获信号，0 表示不反相，1 表示反相。

CC4NP～CC1NP：捕获/比较通道 x 的输出极性，配置为输出时，该位无效，保持为 0；配置为输入时，与 CCxP 组合表示 TI1FP1 和 TI2FP1 的极性和触发电平：00 表示 TIxFP1 的上升沿，不反向，01 表示 TIxFP1 的下降沿，反向，11 表示双沿，反向，10 保留。

CC4E～CC1E：输入捕获通道 x 的输出使能，通道配置为输出时，该位配置为 0 表示禁止 OCx 输出；通道 x 配置为输入时，该位配置为 0 表示禁止捕获。

基本定时器 TIM6 和 TIM7 没有此寄存器，TIM9 只使用低 8 位，TIM10 和 TIM11 只有 CC1NP、CC1P 和 CC1E 域。

10. 计数器 TIMx_CNT

TIMx_CNT 寄存器如图 7-35 所示，16 位的 CNT 表示计数器值。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNT[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 7-35　计数器寄存器

11. 预分频器 TIMx_PSC

TIMx_PSC 用于预分频控制，其寄存器如图 7-36 所示，16 位的 PSC 表示预分频值，计数器的时钟频率 CK_CNT 等于 fCK_PSC/(PSC[15：0]+1)。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PSC[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 7-36　预分频器寄存器结构

12. 自动重装载寄存器 TIMx_ARR

TIMx_ARR 寄存器如图 7-37 所示，有效域 16 位，表示 ARR 自动重装载值，当自动重装载的值为空时，计数器不工作。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ARR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 7-37　自动重装载寄存器

13. 捕获/比较寄存器 TIMx_CCR1、TIMx_CCR2、TIMx_CCR3、TIMx_CCR4

捕获/比较寄存器 TIMx_CCRx 如图 7-38 所示，有效域 CCRx 表示捕获/比较 1 的值，若 CC1 通道配置为输出，CCR1 包含了装入当前捕获/比较 1 寄存器的值(预装载值)，如果在 TIMx_CCMR1 寄存器 OCxPE=0，写入改寄存器的值会被立即传输至当前寄存器中，否则只有当更新事件发生时，才传输至当前当前寄存器中。当前捕获/比较寄存器与计数器 TIMx_CNT 比较，并在 OC1 端口上产生输出信号。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CCR1[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 7-38　捕获/比较寄存器 1

若 CCx 通道配置为输入，CCRx 为上一次输入捕获事件(ICx)记录的计数器值。

TIM6 和 TIM7 没有此寄存器，TIM9 有 CCR1 和 CCR2 两个寄存器，TIM10、TIM11 只有 CCR1 一个寄存器。

7.7　外围定时器库函数

为了便于对通用和基本定时器进行操作，CMSIS 提供了定时器的寄存器定义和库函数。

定时器的寄存器结构体定义 stm32l1xx.h 头文件中，我们可以通过结构体类型 TIM_TypeDef 定义变量 TIMx 对定时器进行控制。

```
typedef struct
{
  uint16_t  CR1;                    //控制寄存器 1
  uint16_t  RESERVED0;
  uint16_t  CR2;                    //控制寄存器 2
  uint16_t  RESERVED1;
  uint16_t  SMCR;                   //从模式控制寄存器
  uint16_t  RESERVED2;
  uint16_t  DIER;                   //中断和 DMA 使能寄存器
  uint16_t  RESERVED3;
  uint16_t  SR;                     //状态寄存器
  uint16_t  RESERVED4;
  uint16_t  EGR;                    //事件产生寄存器
  uint16_t  RESERVED5;
  uint16_t  CCMR1;                  //捕获/比较模式寄存器 1
  uint16_t  RESERVED6;
  uint16_t  CCMR2;                  //捕获/比较模式寄存器 2
  uint16_t  RESERVED7;
  uint16_t  CCER;                   //捕获/比较使能寄存器
  uint16_t  RESERVED8;
  uint32_t  CNT;                    //计数寄存器
  uint16_t  PSC;                    //预分频寄存器
  uint16_t  RESERVED10;
  uint32_t  ARR;                    //自动重装载寄存器
  uint32_t  RESERVED12;
  uint32_t  CCR1;                   //捕获/比较寄存器 1
```

```
    uint32_t  CCR2;                    //捕获/比较寄存器 2
    uint32_t  CCR3;                    //捕获/比较寄存器 3
    uint32_t  CCR4;                    //捕获/比较寄存器 4
    uint32_t  RESERVED17;
    uint16_t  DCR;                     //DMA 控制寄存器
    uint16_t  RESERVED18;
    uint16_t  DMAR;                    //DMA 地址寄存器
    uint16_t  RESERVED19;
    uint16_t  OR;                      //可选功能寄存器
    uint16_t  RESERVED20;
} TIM_TypeDef;
```

CMSIS 提供的主要库函数如表 7-5 所示。

表 7-5　TIM 操作库函数

函　数　名	描　　述
TIM_DeInit	将外设 TIMx 寄存器重设为默认值
TIM_TimeBaseInit	根据 TIM_TimeBaseInitStruct 中指定的参数初始化 TIMx 的时基单元
TIM_TimeBaseStructInit	根据默认值初始化 TimneBaseInitStruct 成员
TIM_OC1Init	根据 TIM_OCInitStruct 中指定的参数初始化外设 TIMx
TIM_OC2Init	根据 TIM_OCInitStruct 中指定的参数初始化外设 TIMx
TIM_OC3Init	根据 TIM_OCInitStruct 中指定的参数初始化外设 TIMx
TIM_OC4Init	根据 TIM_OCInitStruct 中指定的参数初始化外设 TIMx
TIM_OCStructInit	根据默认值初始化 TIM_OCInitStruct 成员
TIM_ICInit	根据 TIM_ICInitStruct 中指定的参数初始化外设 TIMx
TIM_PWMIConfig	根据 TIM_ICInitStruct 中的参数配置外部 PWM 测量
TIM_Cmd	使能或者失能 TIMx 外设
TIM_ITConfig	使能或者失能指定的 TIM 中断
TIM_ETRConfig	配置 TIMx 外部触发
TIM_SelectInputTrigger	选择 TIMx 输入触发源
TIM_PrescalerConfig	设置 TIMx 预分频
TIM_CounterModeConfig	设置 TIMx 计数器模式
TIM_ARRPreloadConfig	使能或失能 TIMx 在 ARR 上的预装载寄存器
TIM_OC1PreloadConfig	使能或失能 TIMx 在 CCR1 上的预装载寄存器
TIM_OC2PreloadConfig	使能或失能 TIMx 在 CCR2 上的预装载寄存器
TIM_OC3PreloadConfig	使能或失能 TIMx 在 CCR3 上的预装载寄存器

续表

函　数　名	描　　述
TIM_OC4PreloadConfig	使能或失能 TIMx 在 CCR4 上的预装载寄存器
TIM_GenerateEvent	设置 TIMx 事件由软件产生
TIM_OC1PolarityConfig	设置 TIMx 通道 1 极性
TIM_OC2PolarityConfig	设置 TIMx 通道 2 极性
TIM_OC3PolarityConfig	设置 TIMx 通道 3 极性
TIM_OC4PolarityConfig	设置 TIMx 通道 4 极性
TIM_SelectOnePulseMode	设置 TIMx 单脉冲模式
TIM_SelectOutputTrigger	选择 TIMx 触发输出模式
TIM_SelectSlaveMode	选择 TIMx 从模式
TIM_SetCounter	设置 TIMx 计数器寄存器值
TIM_SetAutoreload	设置 TIMx 自动重装载寄存器值
TIM_SetCompare1	设置 TIMx 捕获/比较 1 寄存器值
TIM_SetCompare2	设置 TIMx 捕获/比较 2 寄存器值
TIM_SetCompare3	设置 TIMx 捕获/比较 3 寄存器值
TIM_SetCompare4	设置 TIMx 捕获/比较 4 寄存器值
TIM_SetIC1Prescaler	设置 TIMx 输入/捕获 1 预分频
TIM_SetIC2Prescaler	设置 TIMx 输入/捕获 2 预分频
TIM_SetIC3Prescaler	设置 TIMx 输入/捕获 3 预分频
TIM_SetIC4Prescaler	设置 TIMx 输入/捕获 4 预分频
TIM_SetClockDivision	设置 TIMx 的时钟分割值
TIM_GetCapture1	获得 TIMx 输入/捕获 1 的值
TIM_GetCapture2	获得 TIMx 输入/捕获 2 的值
TIM_GetCapture3	获得 TIMx 输入/捕获 3 的值
TIM_GetCapture4	获得 TIMx 输入/捕获 4 的值
TIM_GetCounter	获得 TIMx 计数器的值
TIM_GetPrescaler	获得 TIMx 预分频值
TIM_GetFlagStatus	检查指定的 TIM 标志位设置与否
TIM_ClearFlag	清除 TIMx 的待处理标志位
TIM_GetITStatus	检查指定的 TIM 中断发生与否
TIM_ClearITPendingBit	清除 TIMx 的中断待处理位

1）TIM_DeInit 函数

TIM_DeInit 函数将重新启动 TIMx 控制器时钟，输入参数为 TIMx

函数原型：void TIM_DeInit(TIM_TypeDef * TIMx)

2）TIM_TimeBaseInit 函数

TIM_TimeBaseInit 函数根据 TIM_TimeBaseInitStruct 中指定的参数初始化 TIMx 的时基单元，输入参数为 TIMx，即所要初始化的定时器，函数原型如下：

```
void TIM_TimeBaseInit (TIM_TypeDef * TIMx, TIM_TimeBaseInitTypeDef *
                       TIM_TimeBaseInitStruct)
```

其中 TIM_TimeBaseInitStruct 结构体包含了 TIMx 时基单元配置参数，其定义如下

```
typedef struct
{
    uint16_t TIM_Period;
    uint16_t TIM_Prescaler;
    uint8_t TIM_ClockDivision;
    uint16_t TIM_CounterMode;
} TIM_TimeBaseInitTypeDef;
```

① TIM_Period 为自动重载寄存器值，取值范围为 0x0000～0xFFFF。

② TIM_Prescaler 为时钟的预分频值，取值范围为 0x0000～0xFFFF。

③ TIM_ClockDivision 为数字滤波时钟分频参数，即寄存器 TIMx_CR1 的 CKD 值，其取值为 TIM_CKD_DIV1、TIM_CKD_DIV2 和 TIM_CKD_DIV4，分别表示不分频、2 分频和 4 分频。

④ TIM_CounterMode 设置计数模式，取值为：

- TIM_CounterMode_Up：TIM 向上计数模式；
- TIM_CounterMode_Down：TIM 向下计数模式；
- TIM_CounterMode_CenterAligned1：TIM 中央对齐模式 1 计数模式；
- TIM_CounterMode_CenterAligned2：TIM 中央对齐模式 2 计数模式；
- TIM_CounterMode_CenterAligned3：TIM 中央对齐模式 3 计数模式。

例如，定时器 ARR 为 65535，16 分频，采样时钟不分频，向上计数的配置代码如下：

```
TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
TIM_TimeBaseStructure.TIM_Period = 0xFFFF;
TIM_TimeBaseStructure.TIM_Prescaler = 0xF;
TIM_TimeBaseStructure.TIM_ClockDivision = 0x0;
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
TIM_TimeBaseInit(TIM2, & TIM_TimeBaseStructure);
```

3）TIM_OC1Init 函数

TIM_OC1Init 根据 TIM_OCInitStruct 中指定的参数初始化外设 TIMx 的输出比较通道 1，其函数原型为：

```
void TIM_OC1Init(TIM_TypeDef* TIMx, TIM_OCInitTypeDef* TIM_OCInitStruct)
```

其中，TIM_OCInitStruct 包含了 TIMx 的比较输出配置信息，其结构体定义如下：

```
typedef struct
{
  uint16_t TIM_OCMode;
  uint16_t TIM_Pulse;
  uint16_t TIM_OCPolarity;
} TIM_OCInitTypeDef;
```

① TIM_OCMode 选择定时器模式，对应于 TIMx_CCMRx 的 OCxM，取值为：

- TIM_OCMode_Timing：冻结模式，输出不起作用；
- TIM_OCMode_Active：匹配输出高电平；
- TIM_OCMode_Inactive：匹配输出低电平；
- TIM_OCMode_Toggle：匹配翻转输出；
- TIM_OCMode_PWM1：脉冲宽度调制模式 1；
- TIM_OCMode_PWM2：脉冲宽度调制模式 2。

② TIM_Pulse 设置捕获/比较寄存器的脉冲数，取值范围为 0x0000～ 0xFFFF。

③ TIM_OutputState 设置是否启用输出比较模式，TIM_OutputState_Enable 表示启用，TIM_OutputState_Disable 表示禁用。

④ TIM_OCPolarity 输出极性，取值 TIM_OCPolarity_High 表示输出信号翻转，TIM_OCPolarity_Low 表示输出信号不翻转。

示例：配置 TIM2 的输出通道 1 位 PWM 模式 1。

```
TIM_OCInitTypeDef TIM_OCInitStructure;
TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
TIM_OCInitStructure.TIM_Pulse = 0x3FFF;
TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;
TIM_OC1Init(TIM2, & TIM_OCInitStructure);
```

TIM_OC2Init、TIMOC3Init 和 TIM_OC4Init 函数与 TIM_OC1Init 函数类似。

4）TIM_ICInit 函数

TIM_ICInit 根据 TIM_ICInitStruct 中指定的参数初始化外设 TIMx 的输入捕获通道，其函数原型为：

```
void TIM_ICInit(TIM_TypeDef* TIMx, TIM_ICInitTypeDef* TIM_ICInitStruct)
```

其中，TIM_ICInitStruct 包含了 TIMx 的输入配置信息，其结构体的定义如下：

```
typedef struct
{
  uint16_t TIM_Channel;
  uint16_t TIM_ICPolarity;
```

```
    uint16_t TIM_ICSelection;
    uint16_t TIM_ICPrescaler;
    uint16_t TIM_ICFilter;
} TIM_ICInitTypeDef;
```

① TIM_Channel 选择通道，TIM_Channel_x 表示使用 TIM 通道 x。

② TIM_ICPolarity 输入的捕获信号沿，TIM_ICPolarity_Rising 为上升沿，TIM_ICPolarity_Falling 为下降沿。

③ TIM_ICSelection 选择输入，其取值如下：

- TIM_ICSelection_DirectTI：TIM 输入 x 与 ICx 对应相连；
- TIM_ICSelection_IndirectTI：TIM 输入 x 与 ICx 不对应，交叉相连。

④ TIM_ICPrescaler 设置输入捕获预分频器，其取值 TIM_ICPSC_DIV1、TIM_ICPSC_DIV2、TIM_ICPSC_DIV3、TIM_ICPSC_DIV4 分别表示 TIM 捕获每 1，2，3，4 个事件执行一次。

⑤ TIM_ICFilter 选择输入比较滤波器，取值范围 0x0～0xF。

示例：

```
TIM_ICInitStructure.TIM_Channel      = TIM_Channel_2;
TIM_ICInitStructure.TIM_ICPolarity  = TIM_ICPolarity_Rising;
TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI;
TIM_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1;
TIM_ICInitStructure.TIM_ICFilter = 0x0;
TIM_ICInit(TIM4, &TIM_ICInitStructure);
```

TIM_ICInit 的实现调用 TIx_Config 函数配置 CCER 和 CCMR1 寄存器的极性、通道和滤波，调用 TIM_SetIC1Prescaler 函数配置 CCMR1 寄存器的捕获事件分频。

5）TIM_Cmd 函数

TIM_Cmd 用来使能或停止定时器 TIMx，其函数原型为：

```
void TIM_Cmd(TIM_TypeDef* TIMx, FunctionalState NewState)
```

参数 NewState 的取值为 ENABLE 或者 DISABLE。

6）TIM _ITConfig 函数

TIM_ITConfig 用于配置 TIMx 的中断，其函数原型为：

```
void TIM_ITConfig(TIM_TypeDef* TIMx, uint16_t TIM_IT, FunctionalState NewState)
```

其中 TIM_IT 为 TIM 中断源，包括更新中断 TIM_IT_Update，捕获比较中断 TIM_IT_CC1～TIM_IT_CC4 和触发中断 TIM_IT_Trigger。

例如，启用 TIM2 的捕获比较通道 1 中断 TIM_ITConfig（TIM2，TIM_IT_CC1，ENABLE）。

7）TIM_ETRConfig 函数

TIM_ETRConfig 用于配置外部触发模式 2 下时钟的极性、预分频和滤波参数，其原

型为：

```
void TIM_ETRConfig(TIM_TypeDef * TIMx, uint16_t TIM_ExtTRGPrescaler, uint16_t TIM_ExtTRGPolarity, uint8_t ExtTRGFilter)
```

8）TIM_PrescalerConfig 函数

TIM_PrescalerConfig 函数用于配置分频系数和是否启用预装载，其函数原型为：

```
void TIM_PrescalerConfig(TIM_TypeDef * TIMx, uint16_t Prescaler,uint16_t TIM_PSCReloadMode)
```

参数 TIM_PSCReloadMode 的取值为：TIM_PSCReloadMode_Update 和 TIM_PSCReloadMode_Immediate，选用后者时，TIM 预分频值即时装入。

9）TIM_ARRPreloadConfig 函数

TIM_ARRPreloadConfig 用于配置是否启用 ARR 寄存器预装载功能，对应于 TIMx_CR1 的 ARPE，其函数原型为：

```
void TIM_ARRPreloadConfig(TIM_TypeDef * TIMx, FunctionalState Newstate)
```

参数 NewState 可以取 ENABLE 或 DISABLE。

10）TIM_OC1PreloadConfig 函数

TIM_OC1PreloadConfig 函数用于配置是否启用 OC1 寄存器的预装载功能，其函数原型为：

```
void TIM_OC1PreloadConfig(TIM_TypeDef * TIMx, uint16_t TIM_OCPreload)
```

TIM_OCPreload 参数的取值为预装载使能 TIM_OCPreload_Enable 和预装载禁用 TIM_OCPreload_Disable。

函数 TIM_OC2PreloadConfig、TIM_OC3PreloadConfig、TIM_OC4PreloadConfig 分别用于配置捕获通道 2～4。

11）TIM_SelectSlaveMode 函数

TIM_SelectSlaveMode 用于配置从模式控制器的参数，其函数原型为：

```
void TIM_SelectSlaveMode(TIM_TypeDef * TIMx, uint16_t TIM_SlaveMode)
```

TIM_SlaveMode 参数的取值包括：

- TIM_SlaveMode_Reset：触发信号（TRGI）的上升沿复位计数器并触发更新；
- TIM_SlaveMode_Gated：当触发信号（TRGI）为高电平计数器时钟使能；
- TIM_SlaveMode_Trigger：计数器在触发（TRGI）的上升沿开始计数；
- TIM_SlaveMode_External1 选中触发（TRGI）的上升沿作为计数器时钟。

12）TIM_SetCounter 函数

TIM_SetCounter 用于设置 TIMx 的计数器寄存器值，原型为：

```
void TIM_SetCounter(TIM_TypeDef * TIMx, uint16_t Counter)
```

13）TIM_ SetAutoreload 函数

TIM_ SetAutoreload 用于设置 TIMx 自动重装载寄存器值，其函数原型为：

```
Void TIM_SetAutoreload(TIM_TypeDef * TIMx, uint16_t TIMAutoreload)
```

14）TIM_SetCompare1 函数

TIM_SetCompare1 函数用于设置 TIMx 捕获比较寄存器 1 的值，其函数原型为：

```
void TIM_SetCompare1(TIM_TypeDef * TIMx, uint16_t Compare1)
```

TIM_SetCompare2、TIM_SetCompare3、TIM_SetCompare4 分别用于设置捕获比较寄存器 2～4。

15）TIM_SetIC1Prescaler 函数

TIM_SetIC1Prescaler 用于设置 TIMx 输入捕获 1 的预分频，其函数原型为：

```
void TIM_SetIC1Prescaler(TIM_TypeDef * TIMx, uint16_t TIM_IC1Prescaler)
```

TIM_IC1Prescaler 的取值为 TIM_ICPSC_DIV1、TIM_ICPSC_DIV2、TIM_ICPSC_DIV4、TIM_ICPSC_DIV8，分别表示 0、2、4、8 个事件触发一次捕获。

函数 TIM_SetIC2Prescaler、TIM_SetIC3Prescaler 和 TIM_SetIC4Prescaler 分别用于输入捕获通道 2～4 的设置。

16）函数 TIM_SetClockDivision

TIM_SetClockDivision 用于设置 TIMx 的采样时钟 fdts 分割值，函数原型为：

```
void TIM_SetClockDivision(TIM_TypeDef * TIMx, uint16_t TIM_CKD)
```

TIM_CKD 的时钟分割值取值为 TIM_CKD_DIV1、TIM_CKD_DIV2、TIM_CKD_DIV4，分别表示 CK_CNT 的 1～4 分频。

17）TIM_GetCapture1 函数

TIM_GetCapture1 用于获取捕获寄存器 1 的值，函数原型为：

```
uint16_t TIM_GetCapture1(TIM_TypeDef * TIMx)
```

函数 TIM_GetCapture2、TIM_GetCapture3 、TIM_GetCapture4 分别表示捕获获取捕获寄存器 2～4 的值。

18）TIM_GetCounter 函数

TIM_GetCounter 用于获取定时器的计数器值，函数原型为：

```
uint16_t TIMCounter = TIM_GetCounter(TIM2);
```

19）TIM_GetPrescale 函数

TIM_GetPrescaler 用于获取定时器的时钟预分频值，函数原型为：

```
uint16_t TIM_GetPrescaler (TIM_TypeDef * TIMx)
```

20）TIM_ GetFlagStatus 函数

TIM_ GetFlagStatus 用于获取 TIMx_SR 寄存器中指定的 TIM 标志位的值，输出结果为 SET(1)或 RESET(0)，其函数原型为：

```
FlagStatus TIM_GetFlagStatus(TIM_TypeDef * TIMx, uint16_t TIM_FLAG)
```

可获取的 TIM_FLAG 包括：

- TIM_FLAG_Update：TIM 更新标志位；
- TIM_FLAG_CCx：TIM 捕获/比较通道 x 标志位；
- TIM_FLAG_Trigger：TIM 触发标志位；
- TIM_FLAG_CCxOF：TIM 捕获/比较通道 x 溢出标志位。

21）TIM_ ClearFlag 函数

TIM_ ClearFlag 用于清除 TIMx_SR 寄存器的标志位，其函数原型为：

```
void TIM_ClearFlag(TIM_TypeDef * TIMx, uint32_t TIM_FLAG)
```

TIM_FLAG 参数的取值与 TIM_ GetFlagStatus 函数相同。

22）TIM_ GetITStatus 函数

TIM_ GetITStatus 用于检查 TIMx 的中断发生与否，其函数原型为：

```
ITStatus TIM_GetITStatus(TIM_TypeDef * TIMx, uint16_t TIM_IT)
```

TIM_IT 是待检查的 TIM 中断源，包括：TIM_IT_Update、TIM_IT_CCx 和 TIM_IT_Trigger。

23）TIM_ ClearITPendingBit 函数

TIM_ ClearITPendingBit 用于清除 TIMx 的中断待处理位，其函数原型为：

```
void TIM_ClearITPendingBit(TIM_TypeDef * TIMx, uint16_t TIM_IT)
```

TIM_IT 的取值与 TIM_GetITStatus 函数相同。

24）TIM_PWMIConfig 函数

TIM_PWMIConfig 用于根据 TIM_ICInitStruct 中的参数配置外部 PWM 输入测量，其函数原型为：

```
void TIM_PWMIConfig(TIM_TypeDef * TIMx, TIM_ICInitTypeDef * TIM_ICInitStruct)
```

7.8　定时器应用例程

7.8.1　定时器寄存器操作案例

【例 7-3】 TIM_TimeBaseInit 函数的寄存机操作实现。

```
void TIM_TimeBaseInit(TIM_TypeDef * TIMx, TIM_TimeBaseInitTypeDef * TIM_TimeBaseInitStruct)
```

```
{
    uint16_t tmpcr1 = 0;
    //获取控制寄存器 CR1 的值
    tmpcr1 = TIMx->CR1;
    //如果是通用定时器,需要配置计数模式 TIM_CR1_DIR 和 TIM_CR1_CMS
    tmpcr1 |= (uint32_t)TIM_TimeBaseInitStruct->TIM_CounterMode;
    //如果是基本定时器,只需配置 TIM_CR1_CKD
    tmpcr1 |= (uint32_t)TIM_TimeBaseInitStruct->TIM_ClockDivision;
    TIMx->CR1 = tmpcr1;                                    //将配置写入 CR1 寄存器
    TIMx->ARR = TIM_TimeBaseInitStruct->TIM_Period ;       //写入预装载值
    //写入时钟分频因子
    TIMx->PSC = TIM_TimeBaseInitStruct->TIM_Prescaler;
    //软件产生一个更新事件,将 ARR 和 PSC 的值立即装入影子寄存器
    TIMx->EGR = TIM_PSCReloadMode_Immediate;
}
```

【例 7-4】 TIM_OC1Init 的寄存器操作。

```
void TIM_OC1Init(TIM_TypeDef * TIMx, TIM_OCInitTypeDef * TIM_OCInitStruct)
{
    uint16_t tmpccmrx = 0, tmpccer = 0;
    //配置前首先禁用 CC1
    TIMx->CCER &= (uint16_t) (~ (uint16_t)TIM_CCER_CC1E);
    //保存 CCER 和 CCMR1 的原始值
    tmpccer = TIMx->CCER;
    tmpccmrx = TIMx->CCMR1;
    // 配置输出比较模式
    tmpccmrx |= TIM_OCInitStruct->TIM_OCMode;
    //选择输出极性
    tmpccer |= TIM_OCInitStruct->TIM_OCPolarity;
    //设置比较输出使能
    tmpccer |= TIM_OCInitStruct->TIM_OutputState;
    //设置比较寄存器值
    TIMx->CCR1 = TIM_OCInitStruct->TIM_Pulse;
    //将配置数据写入 CCMR1 寄存器和 CCER 寄存器
    TIMx->CCMR1 = tmpccmrx;
    TIMx->CCER = tmpccer;
}
```

7.8.2 基本计时中断示例

配置一个定时器的基本流程是：

(1) 配置时钟(一般在 systeminit 中已经进行了配置)；

（2）配置中断向量 NVIC_Init()；

（3）配置定时器参数 TIM_TimeBaseInit()，如果不配置，则按照默认参数 ARR＝65535，不分频，向上计数配置；

（4）开启定时器 TIM_Cmd(TIMx,ENABLE)；

（5）中断处理函数 TIMx_IRQHandler()。

【例 7-5】 配置一个通用定时器，使用溢出中断控制 LED 灯，使得 LED 灯以 500ms 的周期翻转。

分析：采用 TIM2 进行配置，TIM2 初始化时主频设定为 32MHz，因此分频设为 3200，这样一个计数为 10kHz，自动重装载值设为 5000，即可得到 500ms 定时长度。在中断产生后，通过状态寄存器的值来判断此次产生的中断属于什么类型。然后执行相关的操作，我们这里使用的是更新中断，在处理完中断之后应该向 TIM2_SR 的最低位写 0，来清除该中断标志。

在固件库函数里面，用来读取中断状态寄存器的值判断中断类型的函数是：ITStatus TIM_GetITStatus，用来判断定时器 TIMx 的中断类型 TIM_IT 是否发生中断。判断定时器 2 是否发生更新中断，方法为：

```
if(TIM_GetITStatus(TIM2,TIM_IT_Update)!=RESET){}
```

清除定时器 TIMx 的中断 TIM_IT 标志位，方法为：

```
TIM_ClearITPendingBit(TIM2,TIM_IT_Update)
```

程序代码如下：

```
void main()
{
    //开启时钟
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2,ENABLE);
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOB,ENABLE);
    //配置中断向量
    NVIC_InitTypeDef NVIC_InitStructure;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
    NVIC_InitStructure.NVIC_IRQChannel=TIM2_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority=0;
    NVIC_InitStructure.NVIC_IRQChannelCmd=ENABLE;
    NVIC_Init(&NVIC_InitStructure);
    //I/O初始化
    GPIO_InitTypeDef GPIO_InitStructure;
    GPIO_InitStructure.GPIO_Pin =GPIO_Pin_7;
    GPIO_InitStructure.GPIO_Speed =GPIO_Speed_40MHz;
    GPIO_InitStructure.GPIO_Mode =GPIO_Mode_Out_PP;
    GPIO_Init(GPIOB,&GPIO_InitStructure);
```

```
    //通用定时器 TIM 设置
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
    TIM_DeInit(TIM2);                                              //缺省 TIMER 配置
    //配置定时器参数
    TIM_ InternalClockConfig (TIM2)                                 //设置 TIM 时钟源为内部
    时钟
    TIM_TimeBaseStructure.TIM_Prescaler=3200-1;                    //设置预分频系数
    TIM_TimeBaseStructure.TIM_ClockDivision=TIM_CKD_DIV1;//设置时钟分频
    //设置计数器计数模式为向上计数
    TIM_TimeBaseStructure.TIM_CounterMode=TIM_CounterMode_Up;
    TIM_TimeBaseStructure.TIM_Period=5000-1;                       //设置定时周期 5000
    TIM_TimeBaseInit(TIM2,&TIM_TimeBaseStructure);                 //初始化设置

    TIM_ClearFlag(TIM2,TIM_FLAG_Update);                           //清除溢出中断标志
    TIM_ ARRPreloadConfig (TIM2, DISABLE)                           //禁止 ARR 预装载缓冲器

    TIM_ITConfig(TIM2,TIM_IT_Update,ENABLE)                        //开启 TIM2 中断
    TIM_Cmd(TIM2,ENABLE);                                          //使能定时器 TIM2
    while(1);
}
```

【**思考题**：上述程序中 TIM_ClearFlag，TIM_ARRPreloadConfig 两行是否可以删掉?】

```
//TIM2 中断处理函数
void TIM2_IRQHandler(void)
{
    Uint8_t ReadValue;
    If(TIM_GetITStatus(TIM2, TIM_IT_Update) !=RESET)
    {
        TIM_ClearITPendingBit(TIM2, TIM_FLAG_Update);          //清除 TIM2 中断
        //读取 PB7 引脚输出数值
        ReadValue ReadValue =GPIO_ReadOutputDataBit(GPIOB,GPIO_Pin_7);
        If(ReadValue ==0)
            GPIO_SetBits(GPIOB,GPIO_Pin_7);
        else
            GPIO_ResetBits(GPIOB,GPIO_Pin_7);
    }
}
```

7.8.3 比较输出示例

比较输出的配置流程为：

(1) 启用 TIM 总线时钟；

（2）配置输出通道的 GPIO 参数，GPIO_Init()、GPIO_PinAFConfig()；
（3）配置时基单元参数；
（4）配置比较输出参数：方向、模式、预装载值、输出极性等；
（5）初始化输出比较寄存器；
（6）配置中断，使能输出比较中断；
（7）启用定时器。

TIM 定时器输入捕获和输出比较的 I/O 引脚如表 7-6 所示。

表 7-6　TIM 对应的输入输出引脚

定时器输入输出通道	I/O 引脚
TIM2_CH1_ETR	PA0、PA5、PA15、PE9
TIM2_CH2	PA1、PB3、PE10
TIM2_CH3	PA2、PB10、PE11
TIM2_CH4	PA3、PB11、PE12
TIM3_ETR	PD2、PE2
TIM3_CH1	PA6、PB4、PC6、PE3
TIM3_CH2	PA7、PB5、PC7、PE4
TIM3_CH3	PB0、PC8
TIM3_CH4	PB1、PC9
TIM4_ETR	PE0
TIM4_CH1	PB6、PD12
TIM4_CH2	PB7、PD13
TIM4_CH3	PB8、PD14
TIM4_CH4	PB9、PD15
TIM9_CH1	PB13、PD0、PE5
TIM9_CH2	PB14、PD7、PE6
TIM10_CH1	PA6、PB8、PB12、PE0
TIM10_CH2	PA7
TIM11_CH1	PB9、PB15、PE1

【例 7-6】　APB1 总线主频设为 8MHz，配置定时器 TIM3，通过定时器 TIM3 的 4 个比较中断通道在 4 个 I/O 口上输出 0.5Hz、1Hz、2Hz 和 4Hz 不同频率的方波，控制 LED 灯。

分析：设定总线主频为 8MHz，此时若采用内部时钟，则驱动定时器的时钟为 16MHz，预分频设为 16000，则一个计数周期为 1ms，这样我们通过给定比较值 1000、500、250 和 125 即可产生 4 个比较中断，通过输出比较通道控制电平进行翻转。

```
#include "stm32L1xx.h"
#include "stm32L1xx_gpio.h"
#include "stm32L1xx_tim.h"
TIM_TimeBaseInitTypeDef   TIM_TimeBaseStructure;
GPIO_InitTypeDef GPIO_InitStructure;
NVIC_InitTypeDef NVIC_InitStructure;
TIM_OCInitTypeDef   TIM_OCInitStructure;
//比较寄存器值
uint16_t CCR1_Val = 1000;
uint16_t CCR2_Val = 500;
uint16_t CCR3_Val = 250;
uint16_t CCR4_Val = 125;
uint16_t PrescalerValue = 0;

int main(void)
{
   //配置总线时钟为 HCLK/4,此时定时器时钟为 HCLK/2 = 16MHz
   RCC_PCLK1Config(RCC_HCLK_Div4);
   //开启 TIM 和 GPIO 时钟
   RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOB, ENABLE);
   RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);
   //TIM3 中断配置
   NVIC_InitStructure.NVIC_IRQChannel = TIM3_IRQn;
   NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
   NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
   NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
   NVIC_Init(&NVIC_InitStructure);
   //GPIO 配置 A6、A7 推挽输出模式,上拉
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6|GPIO_Pin_7 ;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
   GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
   GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;
   GPIO_Init(GPIOA, &GPIO_InitStructure);
   GPIO_PinAFConfig(GPIOA, GPIO_PinSource6, GPIO_AF_TIM3);
   GPIO_PinAFConfig(GPIOA, GPIO_PinSource7, GPIO_AF_TIM3);
   //GPIO 配置 B0 和 B1 推挽输出模式
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0|GPIO_Pin_1 ;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
   GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
   GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;
   GPIO_Init(GPIOB, &GPIO_InitStructure);
```

```
    GPIO_PinAFConfig(GPIOB, GPIO_PinSource0, GPIO_AF_TIM3);
    GPIO_PinAFConfig(GPIOB, GPIO_PinSource1, GPIO_AF_TIM3);
    //时基参数配置
    PrescalerValue = 16000 - 1;
    TIM_TimeBaseStructure.TIM_Period = 65535;
    TIM_TimeBaseStructure.TIM_Prescaler = PrescalerValue;
    TIM_TimeBaseStructure.TIM_ClockDivision = 0x0;
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
    TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure);
    //输出比较通道 1设置
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_Toggle;
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_Low;
    TIM_OCInitStructure.TIM_Pulse = CCR1_Val;
    TIM_OC1Init(TIM3, &TIM_OCInitStructure);
    TIM_OC1PreloadConfig(TIM3, TIM_OCPreload_Disable);
    //输出比较通道 2设置
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_Toggle;
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_Low;
    TIM_OCInitStructure.TIM_Pulse = CCR2_Val;
    TIM_OC2Init(TIM3, &TIM_OCInitStructure);
    TIM_OC2PreloadConfig(TIM3, TIM_OCPreload_Disable);
    //输出比较通道 3设置
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_Toggle;
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_Low;
    TIM_OCInitStructure.TIM_Pulse = CCR3_Val;
    TIM_OC3Init(TIM3, &TIM_OCInitStructure);
    TIM_OC3PreloadConfig(TIM3, TIM_OCPreload_Disable);
    //输出比较通道 4设置
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_Toggle;
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_Low;
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
    TIM_OCInitStructure.TIM_Pulse = CCR4_Val;
    TIM_OC4Init(TIM3, &TIM_OCInitStructure);
    TIM_OC4PreloadConfig(TIM3, TIM_OCPreload_Disable);
    //输出比较中断使能
    TIM_ITConfig(TIM3,TIM_IT_CC1|TIM_IT_CC2|TIM_IT_CC3|TIM_IT_CC4, ENABLE);
    TIM_Cmd(TIM3, ENABLE);                                   //启动定时器
    while (1);
}
```

stm32l1xx_it.c 中断函数

```
uint16_t capture = 0;
extern uint16_t CCR1_Val;
extern uint16_t CCR2_Val;
extern uint16_t CCR3_Val;
extern uint16_t CCR4_Val;

void TIM3_IRQHandler(void)
{
   if (TIM_GetITStatus(TIM3, TIM_IT_CC1) != RESET)
   { //比较中断 1,翻转 PA6,重设比较值
     TIM_ClearITPendingBit(TIM3, TIM_IT_CC1);
     capture = TIM_GetCapture1(TIM3);
     TIM_SetCompare1(TIM2, capture + CCR1_Val);
   }
   else if (TIM_GetITStatus(TIM3, TIM_IT_CC2) != RESET)
   { //比较中断 2,翻转 PA7,重设比较值
     TIM_ClearITPendingBit(TIM3, TIM_IT_CC2);

     capture = TIM_GetCapture2(TIM3);
     TIM_SetCompare2(TIM3, capture + CCR2_Val);
   }
   else if (TIM_GetITStatus(TIM3, TIM_IT_CC3) != RESET)
   { //比较中断 3,翻转 PB0,重设比较值
     TIM_ClearITPendingBit(TIM3, TIM_IT_CC3);
     capture = TIM_GetCapture3(TIM2);
     TIM_SetCompare3(TIM3, capture + CCR3_Val);
   }
   else
   { //比较中断 4,翻转 PB1,重设比较值
     TIM_ClearITPendingBit(TIM3, TIM_IT_CC4);
     capture = TIM_GetCapture4(TIM3);
     TIM_SetCompare4(TIM3, capture + CCR4_Val);
   }
}
```

7.8.4 输入捕获示例

输入捕获的配置流程如下：

(1) 配置 TIM 时钟；

(2) 配置 TIM 输入端口的 GPIO 参数 GPIO_Init()、GPIO_PinAFConfig()；

（3）配置时基参数 ARR、PSR，计数模式、采样频率 TIM_TimeBaseInit()；

（4）配置输入通道参数通道号、极性、捕获模式、分频以及滤波 TIM_ICInit()；

（5）配置 NVIC 中断和 DMA ：NVIC_Init()；

（6）启动定时器 TIM_Cmd(TIMx，ENABLE)；

（7）中断处理中读取捕获值 TIM_GetCapturex()。

【例 7-7】 利用 TIM3 产生一个 1kHz 固定频率的比较输出，将 TIM3 的输出通道通过外部引脚连接到 TIM4 的输入引脚进行二次捕获，然后计算频率，频率＝计数器频率/(两次捕获之间的计数器差值)，由于计数器会溢出，因此可能存在第二次捕获的捕获寄存器值小于第一次，此时需要在结果上加上预装载寄存器 ARR 的值。TIM4 输入捕获的配置代码如下：

```
void TIM_Config(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;
    TIM_ICInitTypeDef   TIM_ICInitStructure;
    //使能 TIM4 和输入通道 PB7 的 GPIO 时钟
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4, ENABLE);
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOB, ENABLE);
    //TIM4 捕获通道 2 的 GPIO 端口配置
    GPIO_InitStructure.GPIO_Mode   = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_PuPd   = GPIO_PuPd_UP;
    GPIO_InitStructure.GPIO_Pin    = GPIO_Pin_7;
    GPIO_Init(GPIOB, &GPIO_InitStructure);
    GPIO_PinAFConfig(GPIOB, GPIO_PinSource7, GPIO_AF_TIM4);
    //TIM4 TI2 输入捕获,上升沿有效,不分频不滤波
    TIM_ICInitStructure.TIM_Channel     = TIM_Channel_2;
    TIM_ICInitStructure.TIM_ICPolarity  = TIM_ICPolarity_Rising;
    TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI;
    TIM_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1;
    TIM_ICInitStructure.TIM_ICFilter = 0x0;
    TIM_ICInit(TIM4, &TIM_ICInitStructure);
    //配置 TIM4 中断
    NVIC_InitStructure.NVIC_IRQChannel = TIM4_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
    //开启定时器,使能通道 2 输入捕获中断,由于没有配置定时器的 ARR 等值,因此采用默认值,即
    //CK_INT 作为时钟源,向上计数,最大计数值为 65535
```

```
    TIM_Cmd(TIM4, ENABLE);
    TIM_ITConfig(TIM4, TIM_IT_CC2, ENABLE);
}

//主程序
int main(void)
{
    //配置 TIM4
    TIM_Config();
    while (1);
    return 0;
}
```

中断处理程序如下：

```
uint16_t IC4ReadValue1 = 0, IC4ReadValue2 = 0;
uint16_t CaptureNumber = 0;
uint32_t Capture = 0;
uint32_t TIM4Freq = 0;
void TIM4_IRQHandler(void)
{
    if (TIM_GetITStatus(TIM4, TIM_IT_CC2) != RESET)
    { //记录第一次捕获中断 CCR 值
      TIM_ClearITPendingBit(TIM4, TIM_IT_CC2);
      if(CaptureNumber == 0)
      {
        IC4ReadValue1 = TIM_GetCapture2(TIM4);
        CaptureNumber = 1;
      }
      else if(CaptureNumber == 1)
      { //记录第二次捕获 CCR 值
        IC4ReadValue2 = TIM_GetCapture2(TIM4);
        //计算差值,如果第二次的 CCR 值小于第一次,则结果加上 65535
        if (IC4ReadValue2 > IC4ReadValue1)
          Capture = (IC4ReadValue2 - IC4ReadValue1) - 1;
        else if (IC4ReadValue2 < IC4ReadValue1)
          Capture = ((0xFFFF - IC4ReadValue1) + IC4ReadValue2) - 1;
        else
            Capture = 0;
        //计算频率,频率=主频/捕获差值
        TIM4Freq = (uint32_t) SystemCoreClock / Capture;
        CaptureNumber = 0;
      }
    }
}
```

7.8.5　PWM 输出和输入示例

PWM 的输出配置和比较输出模式一样，区别在于输出比较模式不同，且必须开启预装载功能。

【例 7-8】 配置 TIM11 输出一个占空比为 50%，频率为 50kHz 的 PWM 波。

分析：设置 TIM11 的时钟为内部时钟 32MHz，预分频为 0，这样定时器的周期＝32M/50k＝640，因此 ARR＝639，占空比＝（TIM11_CCR1/（TIM11_ARR ＋ 1）），因此 CCR1 设为 320 即可产生一个 50%占空比的方波，代码如下：

```
TIM_TimeBaseInitTypeDef   TIM_TimeBaseStructure;
GPIO_InitTypeDef GPIO_InitStructure;
TIM_OCInitTypeDef   TIM_OCInitStructure;
uint16_t CCR1Val = 320;
int main(void)
{
   //使能定时器和输出引脚 GPIO 时钟
   RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM11, ENABLE);
   RCC_AHBPeriphClockCmd( RCC_AHBPeriph_GPIOB, ENABLE);
   //配置 PWM 输出引脚的 I/O 参数,并映射到 TIM11 的输出通道 1
   GPIO_InitStructure.GPIO_Mode   = GPIO_Mode_AF;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;
   GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
   GPIO_InitStructure.GPIO_PuPd   = GPIO_PuPd_UP;
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
   GPIO_Init(GPIOB, &GPIO_InitStructure);
   GPIO_PinAFConfig(GPIOB, GPIO_PinSource9, GPIO_AF_TIM11);
   //TIM11 时基单元参数配置
   TIM_TimeBaseStructure.TIM_Period = 639;
   TIM_TimeBaseStructure.TIM_Prescaler = 0;
   TIM_TimeBaseStructure.TIM_ClockDivision = 0;
   TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
   TIM_TimeBaseInit(TIM11, &TIM_TimeBaseStructure);

   //配置输出通道 1 为 PWM1 模式
   TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
   TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
   TIM_OCInitStructure.TIM_Pulse = CCR1Val;
   TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;
   TIM_OC1Init(TIM11, &TIM_OCInitStructure);
   TIM_OC1PreloadConfig(TIM11, TIM_OCPreload_Enable);
   TIM_ARRPreloadConfig(TIM11, ENABLE);
```

```
    //启动定时器
    TIM_Cmd(TIM11, ENABLE);
    while (1){}
}
```

【例 7-9】 利用上例中的 TIM11 输出的 PWM，将其输出引脚接到 TIM3 的输入通道 2 上，对输入 PWM 进行频率和占空比测量。

分析：TIM3 的通道 1 和通道 2 同时对 PWM 信号进行捕获，频率＝TIM3 频率/TIM3_CCR2 的值，占空比＝（TIM3_CCR1 * 100)/(TIM3_CCR2)。

在 PWM 输入模式下，输入配置需要使用 TIM_PWMIConfig 函数对 TIM_ICInitStructure 进行配置。代码如下：

```
TIM_ICInitTypeDef TIM_ICInitStructure;
GPIO_InitTypeDef GPIO_InitStructure;
NVIC_InitTypeDef NVIC_InitStructure;
int main(void)
{
    //TIM3 和输入捕获引脚 GPIO 时钟开启
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);
    //PA7 捕获引脚 GPIO 参数配置
    GPIO_InitStructure.GPIO_Pin =   GPIO_Pin_7;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_UP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource7, GPIO_AF_TIM3);
    // TIM3 中断配置
    NVIC_InitStructure.NVIC_IRQChannel = TIM3_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
    //TIM3 PWM 输入模式配置,通道 2,上升沿,CCR2 计算频率,CCR1 计算占空比,
    //无分频、无滤波
    TIM_ICInitStructure.TIM_Channel = TIM_Channel_2;
    TIM_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Rising;
    TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI;
    TIM_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1;
    TIM_ICInitStructure.TIM_ICFilter = 0x0;
    //使用 PWMIConfig 函数初始化
    TIM_PWMIConfig(TIM3, &TIM_ICInitStructure);
```

```
    //选择触发源为 TI2
    TIM_SelectInputTrigger(TIM3, TIM_TS_TI2FP2);
    //选择从模式控制为复位模式
    TIM_SelectSlaveMode(TIM3, TIM_SlaveMode_Reset);
    //使能主从模式
    TIM_SelectMasterSlaveMode(TIM3, TIM_MasterSlaveMode_Enable);
    TIM_Cmd(TIM3, ENABLE);                                  //启用定时器
    TIM_ITConfig(TIM3, TIM_IT_CC2, ENABLE);                 //开启 TIM3 捕获 2 中断
    while (1);
}
```

中断处理函数为：

```
void TIM3_IRQHandler(void)
{
    //清除捕获中断
    TIM_ClearITPendingBit(TIM3, TIM_IT_CC2);
    //读取通道 1 和通道 2 的捕获值
    IC2Value = TIM_GetCapture2(TIM3);
    if (IC2Value != 0)
    {
      //占空比计算
       DutyCycle = (TIM_GetCapture1(TIM3)  *  100)  / IC2Value;
      //频率计算
       Frequency = SystemCoreClock / IC2Value;
    }
    else
    {
       DutyCycle = 0;
       Frequency = 0;
    }
}
```

Chapter 8

第 8 章　USART 串口控制器

【导读】 USART 是嵌入式系统中串行通信的常用部件,本章首先介绍串行输入输出的基本概念,同步串行和异步串行各自的特点,然后对 STM32L152 的 USART 串行接口控制器的内部结构,寄存器以及发送和接收配置流程进行介绍,最后介绍 CMSIS 提供的典型寄存器操作库函数。针对串口数据帧的传输,本章还对 HDLC 链路协议的帧构造,Modubs 通信进行介绍,以异步串口 PC 连接通信、状态机数据发送和接收为案例介绍 USART 库函数的使用方法。

8.1　串行输入输出接口的基本概念

计算机与外部数据交互的方法有并行传输和串行传输两种,并行传输一次可以传输多个二进制位,需要多根数据线同时进行传输,串行传输一次只能传输一个二进制位,数据需要一位一位地传输。一般来讲,并行传输的吞吐量高,例如计算机系统的内存接口总线等,但并行传输对线路之间的抗干扰能力要求较高,当时钟速率越高,线之间的距离越近时,线间的串扰和线上的延迟对数据传输性能都会产生影响,占用的微处理器口线也较多。而串行传输时,线路比较简单,出错时只需要传输若干位即可,适合远距离传输,且在实现上更容易提高时钟速率,获得较好的通信带宽,因此目前计算机系统里使用的 USB、SPI、SATA、网卡等高速设备均采用串行总线进行通信。

串行数据的传输分为三种模式:

- 单工传输:单工传输下一个设备只能发送或者接收;
- 半双工传输:半双工模式下,通信双方都可以进行发送和接收,但一台设备不能同时发送和接收;
- 双工传输:双工模式下传输效率最高,可以同时进行双向通信。

数字传输用 0 和 1 表示信号,传输距离比较短,且容易受到干扰,因此在数据传输时一般将数字型号调制成模拟信号,在接收端再将模拟信号解调成数字信号,这个功能由调制解调器完成,调制方法包括调幅 ASK、调频 FSK、调相 PSK 等,通过调制,一方面可以用一个模拟符号表示多个二进制位从而提高数据传输速率,另外还可以通过纠错编码提高传输可靠性。调制完的模拟信号具有不同的电气特征。以串行接口的电气标准分,串行接口包括 RS-232-C、RS-422、RS485、USB 等。RS-232-C、RS-422 与 RS-485 标准只对接口的电气特

性做出规定，不涉及接插件、电缆或协议。USB 是近几年发展起来的新型接口标准，主要应用于高速数据传输领域。

RS-232-C：也称标准串口，是目前最常用的一种串行通信接口。它是在 1970 年由美国电子工业协会(EIA)联合调制解调器、计算机终端生产厂家共同制定的用于串行通讯的标准。它的全名是“数据终端设备(DTE)和数据通信设备(DCE)之间串行二进制数据交换接口技术标准”。传统的 RS-232-C 接口标准有 22 根线，采用标准 25 芯 D 型插头座。自 IBM PC/AT 开始使用简化了的 9 芯 D 型插座，RS-232-C 曾是 PC 的标配接口，目前已被 PC 淘汰。

RS-422：为改进 RS-232 通信距离短、速率低的缺点，RS-422 定义了一种平衡通信接口，将传输速率提高到 10Mb/s，传输距离延长到 4000 英尺(速率低于 100kb/s 时)，并允许在一条平衡总线上连接最多 10 个接收器。RS-422 是一种单机发送、多机接收的单向、平衡传输规范，被命名为 TIA/EIA-422-A 标准。

RS-485：为扩展应用范围，EIA 又于 1983 年在 RS-422 基础上制定了 RS-485 标准，增加了多点、双向通信能力，即允许多个发送器连接到同一条总线上，同时增加了发送器的驱动能力和冲突保护特性，扩展了总线共模范围，后命名为 TIA/EIA-485-A 标准。

Universal Serial Bus(通用串行总线 USB)：USB 接口是电脑主板上的一种四针接口，其中中间两个针传输数据，两边两个针给外设供电。USB 接口速度快、连接简单、不需要外接电源，传输速度 12Mb/s，USB 2.0 可达 480Mb/s；电缆最大长度 5 米，USB 电缆有 4 条线：2 条信号线，2 条电源线，可提供 5V 电源；USB 通过串联方式最多可串接 127 个设备；支持热插拔。最新的规格是 USB 3.1。因此在嵌入式系统 RS-232 串口与 PC 通信时，一般采用 USB 转串口的设备进行转接。

RJ-45 接口：是以太网最为常用的接口，RJ-45 是一个常用名称，指的是由 IEC(60)603-7 标准化，使用由国际性的接插件标准定义的 8 个位置(8 针)的模块化插孔或者插头。

计算机的串行总线控制器输入和输出是数字信号，因此在使用 RS-232、RS-485 或 RJ-45 等接口时，需要外加物理层芯片进行信号转换(如 MAX232)。例如，TTL 高电平 1，电压≥2.4V，低电平 0，电压≤0.5V(对于 5V 或 3.3V 电源电压)；RS-232 采用的是负逻辑，高电平 1，电压－15V～－3V，低电平 0，电压＋3V～＋15V；TTL 电平以电源为参考，高电平 1，电压≥0.7 * VCC，低电平 0，电压≤0.2 * VCC。

8.2　串行通信协议

串行数据传输主要采用两种传输协议：同步传输协议和异步传输协议。

8.2.1　异步串行通信协议

异步通信是我们最常采用的通信方式，异步通信采用固定的通信格式，数据以相同的帧

格式传送。如图 8-1 所示,每一帧由起始位、数据位、奇偶校验位和停止位组成。

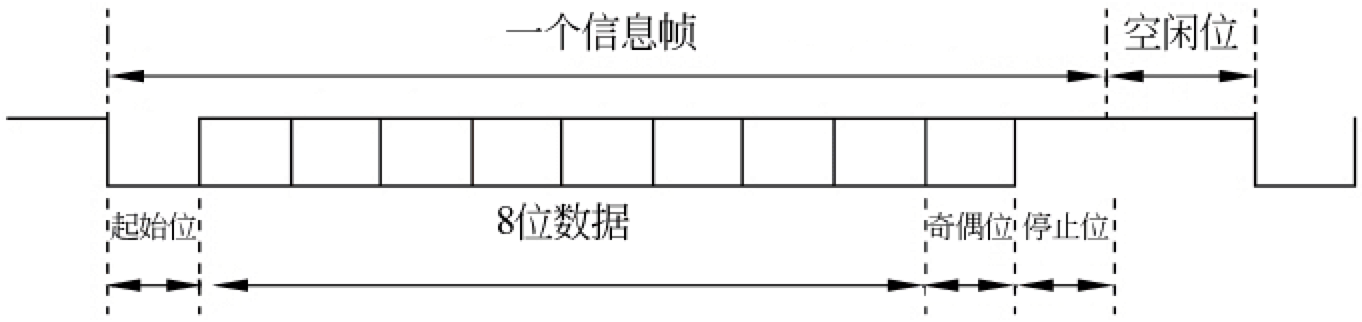


图 8-1 异步串行数据格式

在通信线上没有数据传送时处于逻辑 1 状态(高电平)。当发送设备发送一个字符数据时,首先发出一个逻辑 0 信号(低电平),这个低电平就是起始位。起始位通过通信线传向接收设备,当接收设备检测到这个低电平后,就开始准备接收数据信号。因此,起始位所起的作用就是表示字符传送开始。起始位后面紧接着的是数据位,它可以是 5 位、6 位、7 位或 8 位。数据传送时,低位在前。奇偶校验位用于数据传送过程中的数据检错,数据通信时通信双方必须约定一致的奇偶校验方式。奇偶校验位是冗余位,可以不要校验位。在奇偶校验位或数据位后紧接的是停止位,停止位可以是 1 位、也可以是 1.5 位或 2 位。接收端收到停止位后,知道上一字符已传送完毕,同时,也为接收下一字符做好准备。若停止位后不是紧接着传送下一个字符,则让线路保持为逻辑 1。逻辑 1 表示空闲,线路处于等待状态。

8.2.2 同步串行通信协议

同步通信时,通信双方共用一个时钟,这是同步通信区分于异步通信的最显著的特点。在异步通信中,每个字符要用起始位和停止位作为字符开始和结束的标志,以致占用了部分时间。所以在数据块传送时,为提高通信速度,常去掉这些标志,而采用同步通信。同步通信中,数据开始传送前用同步字符(通常为 1～2 个特殊字符)来指示,并由时钟来实现发送端和接收端的同步,即检测到规定的同步字符后,下面就连续按顺序传送数据,直到一块数据传送完毕。同步传送时,字符之间没有间隙,也不要起始位和停止位,仅在数据开始时用同步字符 SYNC 来指示,其数据格式如图 8-2 所示。

图 8-2 同步串行数据格式

同步通信和异步通信相比,以同步字符作为传送的开始,从而使收发双方取得同步,每位占用的时间相等,字符数据之间不允许有空位,当线路空闲或没字符可发时,发送同步字符。在同步传送时,要求用时钟来实现发送端和接收端之间的同步。为了保证接收正确无误,发送方除了传送数据外,还要传送同步时钟。同步串行通信虽然可以提高传送速度,但实现起来较为复杂。

8.2.3　串行通信基本概念

1. 波特率

波特率(Baud Rate)是指数据传送时,每秒传送数据二进制代码的位数,它的单位是位/秒(b/s)。1 波特就是一位每秒。假设数据传送速率是每秒 120 个字符,而每个字符格式包括 10 个二进制位(1 个起始位、一个终止位、8 个数据位),这时传送的波特率为:10×120=1200b/s。位传送时间宽度 Td=波特率的倒数,则上式中的 Td=1/1200s=0.883ms。

在异步串行通信中,接收设备和发送设备无需传输时钟,但双方必须用各自的时钟保持相同的传送波特率,并以每个字符数据的起始位与发送设备保持同步。起始位、数据位、奇偶位和停止位的约定,在同一次传送过程中必须保持一致,这样才能成功的传送数据。

2. 接收/发送时钟

二进制数据系列在串行传送过程中以数字信号波形的形式出现。不论接收还是发送,都必须有时钟信号对传送的数据进行定位。接收/发送时钟就是用来控制通信设备接收/发送字符数据速度的,该时钟信号通常由外部时钟电路产生。

在发送数据时,发送器在发送时钟的下降沿将移位寄存器的数据串行移位输出;在接收数据时,接收器在接收时钟的上升沿对接收数据采样,进行数据位检测,接收/发送时钟频率与波特率有如下关系:收/发时钟频率=n×收/发波特率,n=1,8,16,64,时钟周期 Tc=Td/n。

在同步传送方式,必须取 n=1,即接收/发送时钟的频率等于收/发波特率。在异步传送方式,n 可配置为 8、16 或 64,即可以选择接收/发送时钟频率是波特率的 1、16、64 倍。因此可由要求的传送波特率及所选择的倍数 n 来确定接收/发送时钟的频率,如图 8-3 所示。

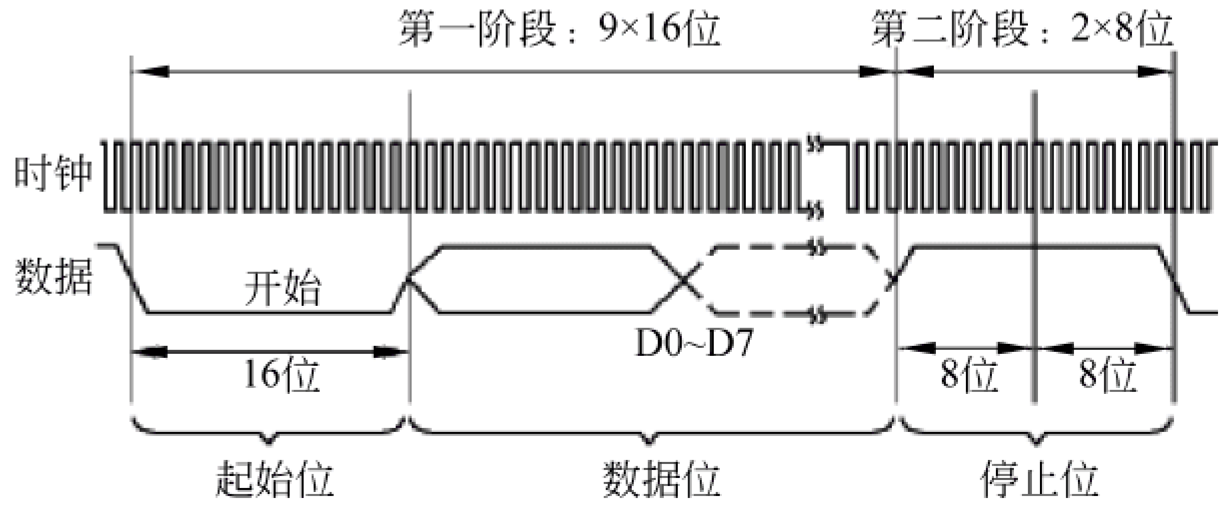


图 8-3　发送和接收时钟及波特率的关系

若取 n=16,那么异步传送接收数据实现同步的过程如下:接收器在每一个接收时钟的上升沿采样接收数据线,当发现接收数据线出现低电平时就认为是起始位的开始,以后若在连续测 8 个时钟周期(因 n=16,故 Td=16 * Tc)内检测到接收数据线仍保持低电平,则确定它为起始位(不是干扰信号)。通过这种方法,不仅能够排除接收线上的噪声干扰,识别假起始位,而且能够相当精确的确定起始位的中间点,从而提供一个正确的时间基准。从这个基准算起,每隔 16 * Tc 采样一次数据线,作为输入数据。一般来说,从接收数据线检测到

一个下降沿开始，若其低电平能保持 $n*Tc/2$（半位时间），则确定为起始位，其后每隔 $n*Tc$ 时间（一个数据时间）在每个数据位的中间点采样，时序如图 8-4 所示。

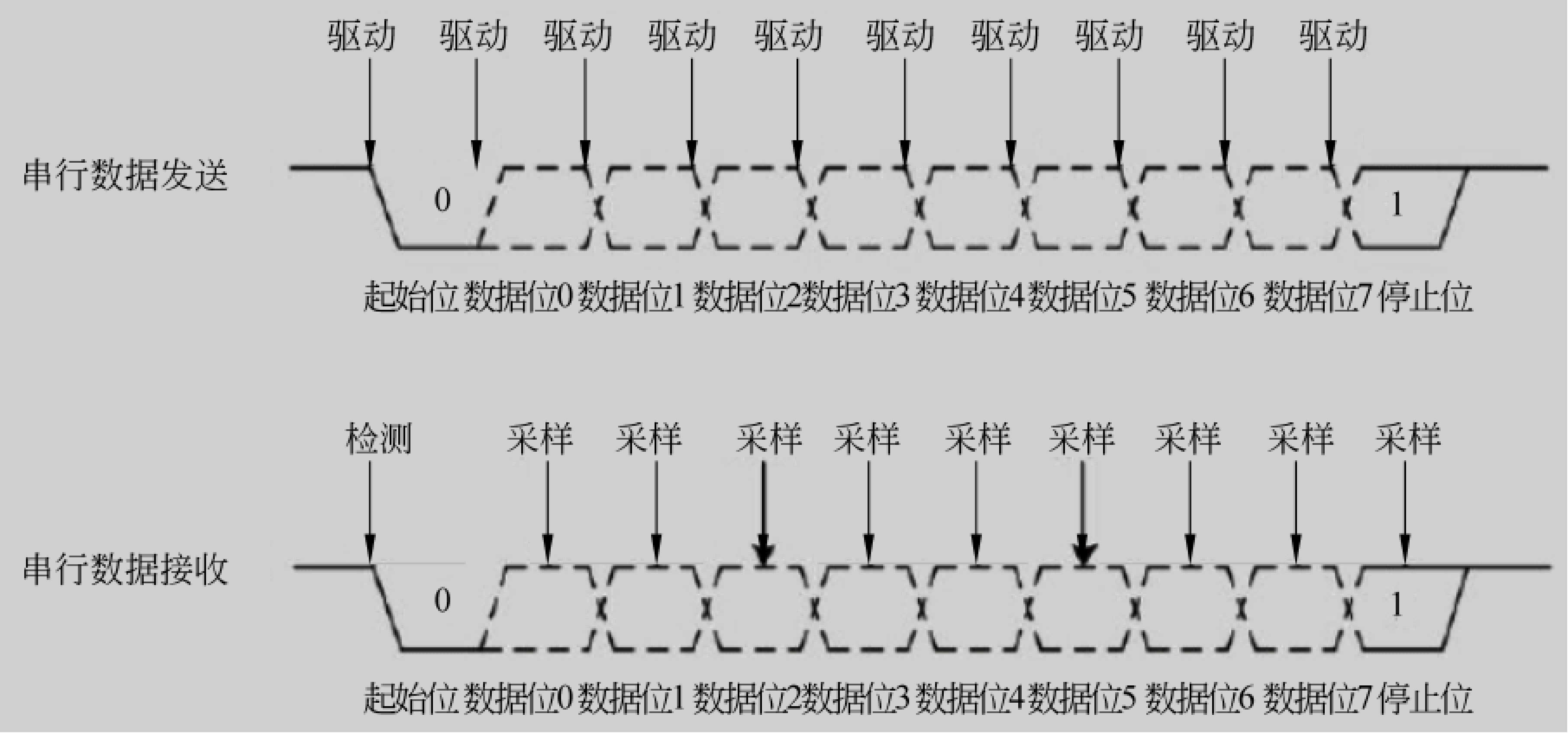


图 8-4　数据发送和采样时钟的驱动时机

3. 数据收发过程

发送数据过程：

- 空闲状态，线路处于高电位；
- 当收到发送数据指令后，拉低线路一个数据位的时间 T；
- 数据按低位到高位依次发送。

数据发送完毕后，接着发送奇偶校验位和停止位（停止位为高电位）。

接收数据过程：

- 空闲状态，线路处于高电位；当检测到线路的下降沿时说明线路有数据传输；
- 按照约定的波特率从低位到高位接收数据；
- 数据接收完毕后，接着接收并比较奇偶校验位是否正确；
- 如果正确则通知后续设备准备接收数据或存入缓存。

4. 空闲帧和断开帧

空闲帧和断开帧如图 8-5 所示。

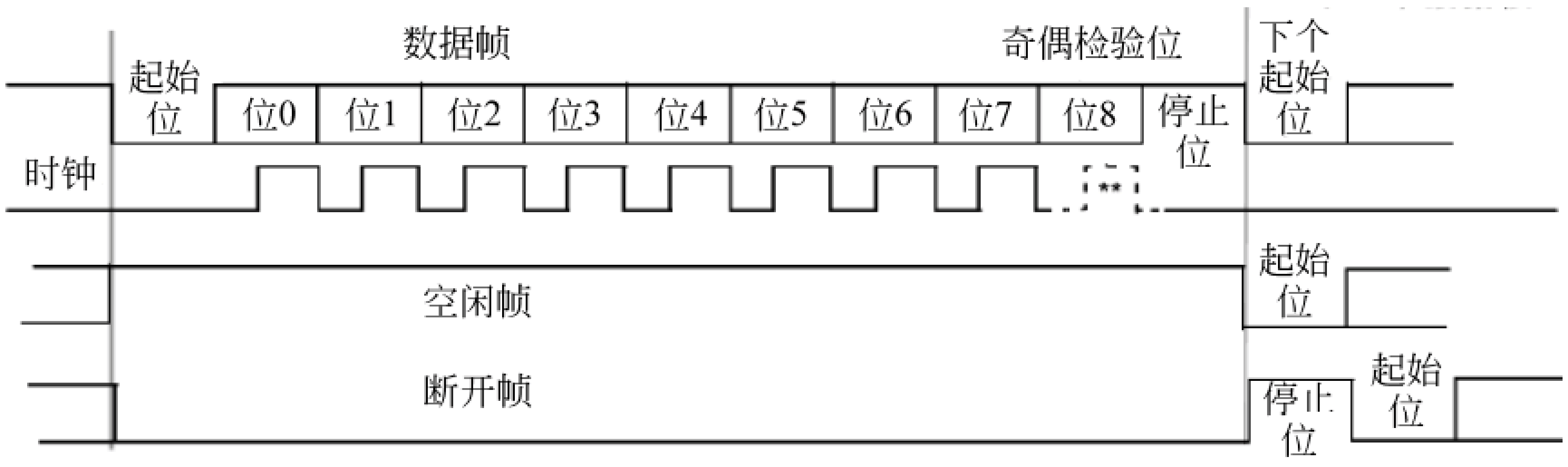


图 8-5　空闲帧和断开帧

- 空闲帧是完全由 1 组成的一个完整的数据帧，后面跟着包含了数据的下一帧的开始位(其中 1 的位数包括了停止位的位数)。
- 断开帧是在一个帧周期内全部收到 0(包括停止位期间也是 0)。在断开帧结束时，发送器可以再插入 1 或 2 个停止位来发起或接收下一个起始位。

5. 硬件流控制

数据在两个串口之间传输时，常常会出现丢失数据的现象，或者两台计算机的处理速度不同，如台式机与单片机之间的通信，接收端数据缓冲区已满，则此时继续发送来的数据就会丢失。流控制能解决这个问题，当接收端数据处理不过来时，就发出“不再接收”的信号，发送端就停止发送，直到收到“可以继续发送”的信号再发送数据。因此流控制可以控制数据传输的进程，防止数据的丢失。两种常用的流控制是硬件流控制和软件流控制。如果 UART 只有 RX、TX 两个信号，要流控的话只能是软件流控，我们在普通的控制通信中一般不用硬件流控制，而用软件流控制。一般通过 xon/xoff 两个特殊字符来实现软件流控制。常用方法是：当接收端的输入缓冲区内数据量超过设定的高位时，就向数据发送端发出特殊字符 xoff (ASCII 码 19，表示 control＋s，也可自定义)，发送端收到 xoff 字符后就立即停止发送数据；当接收端的输入缓冲区内数据量低于设定的低位时，就向数据发送端发出特殊字符 xon (ASCII 码 17 或 control＋q，也可自定义)，发送端收到 xon 字符后就立即开始发送数据。但是在二进制数据传输中，标志字符也有可能在数据流中出现从而引起误操作，因此可以采用硬件流控制解决。硬件流控制常用的有 RTS/CTS (请求发送/清除发送)流控制和 DTR/DSR(数据终端就绪/数据设置就绪)流控制，常用的方式是 RTS/CTS 硬件流控制。

【**思考题**：如何解决软件流控制中传输的数据内部出现 xon 或 xoff 字符的问题?】

RTS(Require To Send，请求发送)为输出信号，用于指示本设备准备好可接收数据，低电平有效，低电平说明本设备可以接收数据，由接收模块向外发出。

CTS(Clear To Send，发送允许)为输入信号，用于判断是否可以向对方发送数据，低电平有效，低电平说明本设备可以向对方发送数据，若是高电平，在当前数据传输结束时阻断下一次的数据发送，由发送模块接收此信号。

两个 USART 设备进行连接时，CTS 和 RTS 进行交叉连接，如图 8-6 所示。如果不使用 USART 的内部硬件流控制模块进行 CTS 和 RTS 控制，我们也可以通过 GPIO 模拟 RTS 和 CTS。

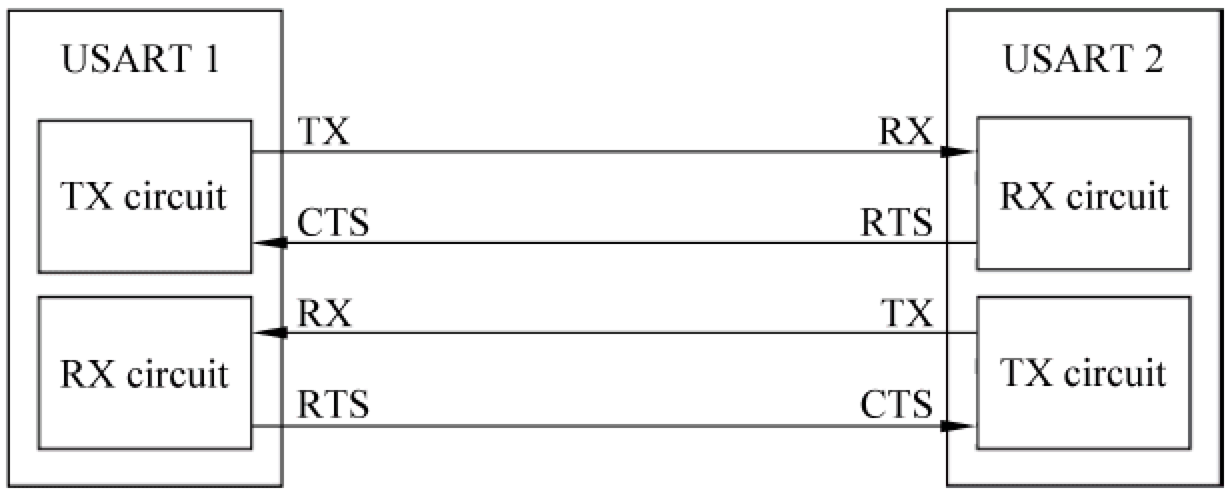


图 8-6　RTS/CTS 硬件流控制连接方式

6. 异步串行通信控制器 UART 的基本结构

如图 8-7 所示，一个异步串行总线控制器由波特率发生器、发送和接收数据控制单元以及串行并行转换单元、中断控制器等几个基本组件组成。波特率发生器用于产生发送和采样时钟，配置数据传输速率，TX 发送单元用于从总线写数据到 UART 进行发送，RX 接收单元用于将接收到的数据传输到总线。串并行转换是两个移位寄存器，用于将一个字符数据逐位发送到 TX 引脚或从 RX 引脚将每个位移位构成一个字符数据。中断控制用于将 UART 传输控制过程中的事件或错误引起的中断发送到 NVIC 控制器。

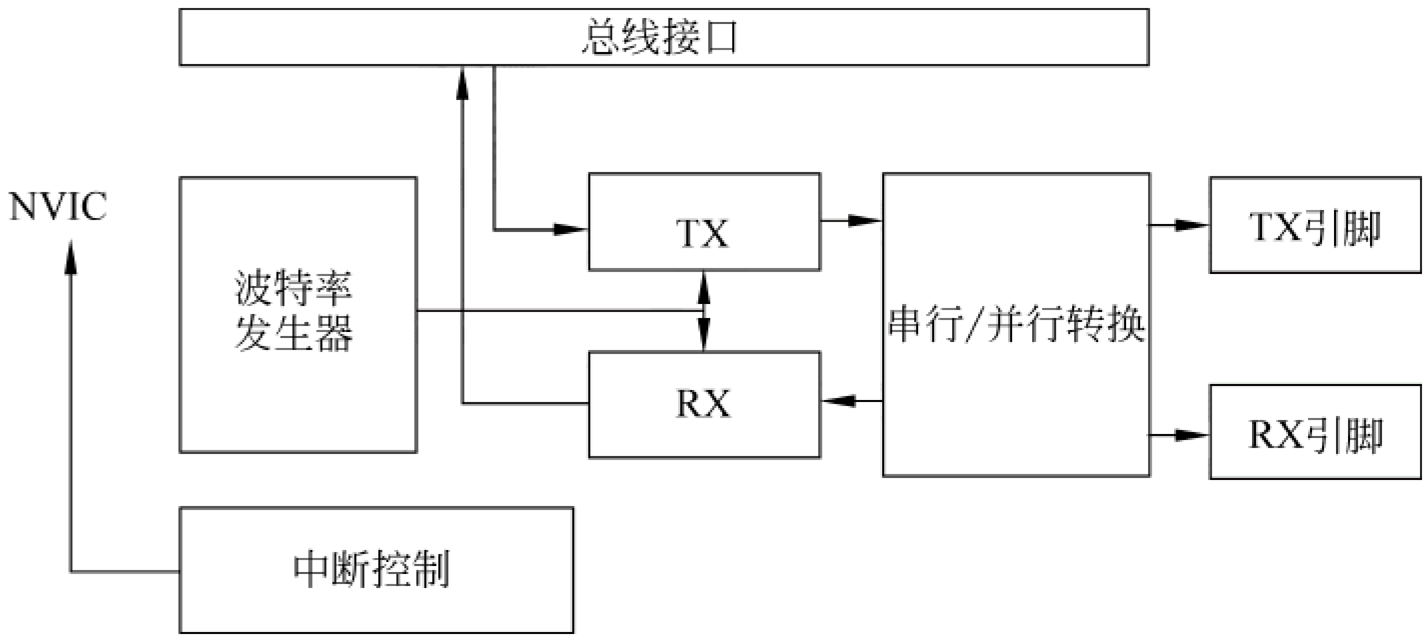


图 8-7　UART 的内部结构

8.3　STM32L152 USART 内部结构与原理

STM32L152 内部集成的通用同步异步收发器(USART)提供了一种灵活的方法与使用工业标准 NRZ 异步串行数据格式的外部设备之间进行全双工数据交换。USART 模式支持：通用全双工异步通信模式(UART)、智能卡模式(ISO7816-3，单线半双工异步模式)、通用全双工同步通信模式(USRT)、硬件流控模式(调制解调器)、IrDA 红外模式、LIN 通信模式。此外它还允许多处理器通信。为实现高速数据通信，可使用多缓冲器配置的 DMA 方式。本章主要对 USART 的异步通信 UART 进行介绍。

STM32L152 系列微控制器最多可集成 5 个串行控制器，每个串行控制器所支持的功能如图 8-8 所示。串口 1～3 是全功能串行控制器，串口 4 和串口 5 不支持同步模式、硬件流控制和智能卡模式。其控制器特点为：

(1) 可编程的波特率发生器系统，最高达 4Mb/s，采样时钟支持 8/16 倍波特率采样时钟；

(2) 可编程数据字长度(8 位或 9 位)，可配置的停止位(支持 1 或 2 个停止位)；

(3) 发送方为同步传输提供时钟；

(4) 支持全双工异步通信和单线半双工通信；

(5) 支持 LIN、SmartCard、IrDA 等多种模式；

(6) 可配置使用 DMA 的多缓冲器通信，在 SRAM 里利用集中式 DMA 缓冲接收/发送

字节；

(7) 单独控制的发送器和接收器使能位；

(8) 支持接收缓冲器满、发送缓冲器空、传输结束标志等多种检测标志，支持发送和接收校验控制；

(9) 支持多种错误检测标志，10 个带标志的中断源；

(10) 支持多处理器通信、静默模式唤醒等。

USART模式	USART1	USART2	USART3	UART4	UART5
异步模式	X	X	X	X	X
硬件流控制	X	X	X	NA	NA
多缓存通信(DMA)	X	X	X	X	X
多处理器通信	X	X	X	X	X
同步	X	X	X	NA	NA
智能卡	X	X	X	NA	NA
半双工(单线模式)	X	X	X	X	X
IrDA	X	X	X	X	X
LIN	X	X	X	X	X

图 8-8　STM32L152 USART 控制器支持的功能列表

一个全功能的串行控制器内部结构如图 8-9 所示。内部模块主要包括发送和接收单元(数据寄存器和移位寄存器)、红外编解码单元、时钟输出控制单元、发送和接收控制单元、硬件流控制单元、波特率控制单元和中断控制单元。内部涉及的寄存器包括控制寄存器 CR、状态寄存器 SR、数据寄存器 TDR、RDR，波特率因子寄存器 BRR 等。

外部引脚包括 TX、RX、IRDA_IN、IRDA_OUT、RTS、CTS 和 CK。对应的 GPIO 引脚如表 8-1 所示。

表 8-1　USART 控制器专用 I/O 引脚

USART 专用功能	UART1 I/O 引脚	UART2 I/O 引脚	UART3 I/O 引脚	UART4 I/O 引脚	UART5 I/O 引脚
TX/ IRDA_OUT	PA9、PB6	PA2、PD5	PB10、PC10、PD8	PC10	PC12
RX/ IRDA_IN	PA10、PB7	PA3、PD6	PB11、PC11、PD9	PC11	PD2
RTS	PA12	PA1、PD4	PB14、PD12		
CTS	PA11	PA0、PD3	PB13、PD11		
CK	PA8	PA4、PD7	PB12、PC12、PD10		

任何 USART 双向通信至少需要两个脚：接收数据输入(RX)和发送数据输出(TX)。当发送器被禁止时，TX 输出引脚恢复为通用 I/O 端口功能。当发送器被激活，并且不发送数据时，TX 引脚处于高电平。在单线和智能卡模式里，此 I/O 口被同时用于数据的发送和接收。

在同步模式中需要使用 CK 引脚，CK 为发送器时钟输出，用于同步传输的时钟，数据

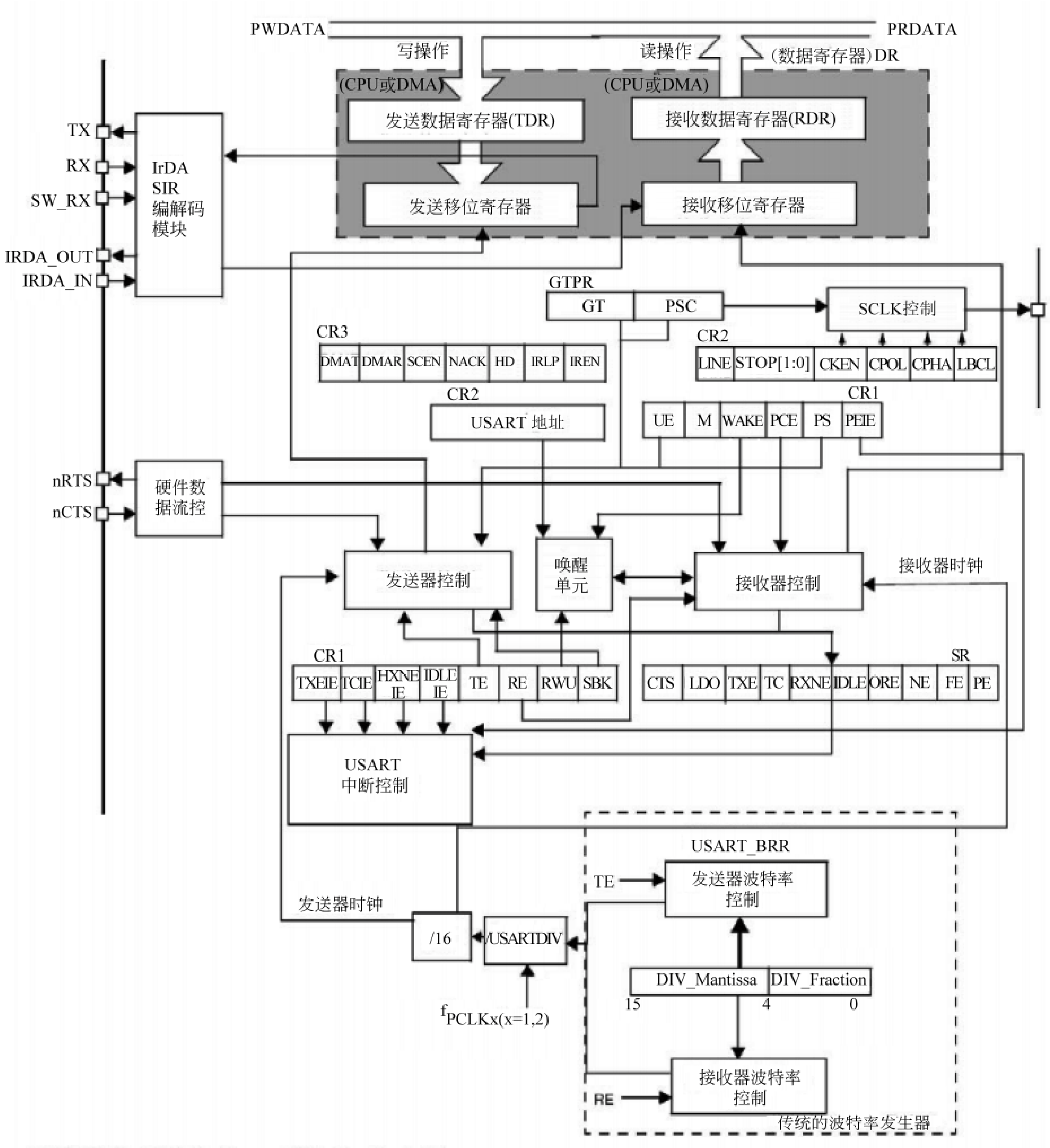


图 8-9　STM32L152 USART 控制器内部结构

可以在 RX 上同步被接收，时钟的相位和极性都是软件可编程的。在智能卡模式里，CK 可以为智能卡提供时钟。在 IrDA 模式里需要使用 IrDA_RDI(IrDA 模式下的数据输入)和 IrDA_TDO(IrDA 模式下的数据输出)引脚。在硬件流控模式中需要使用 nCTS 和 nRTS 引脚。

8.3.1　发送器

发送器发送 8 位或 9 位的数据字，字长可以通过编程 USART_CR1 寄存器中的 M 位来选择成 8 或 9 位。当发送使能位(TE)被设置时，发送移位寄存器中的数据在 TX 脚上输

出，相应的时钟脉冲在 CK 脚上输出。

1. 字符发送

在 USART 发送期间，在 TX 引脚上首先移出数据的最低有效位。对 USART_DR 寄存器的写操作实际上是对发送数据寄存器 TDR 的操作。每个字符之前都有一个低电平的起始位，之后跟着的停止位，停止位的数目可配置，USART 支持 0.5、1、1.5 和 2 个停止位，如图 8-10 所示。

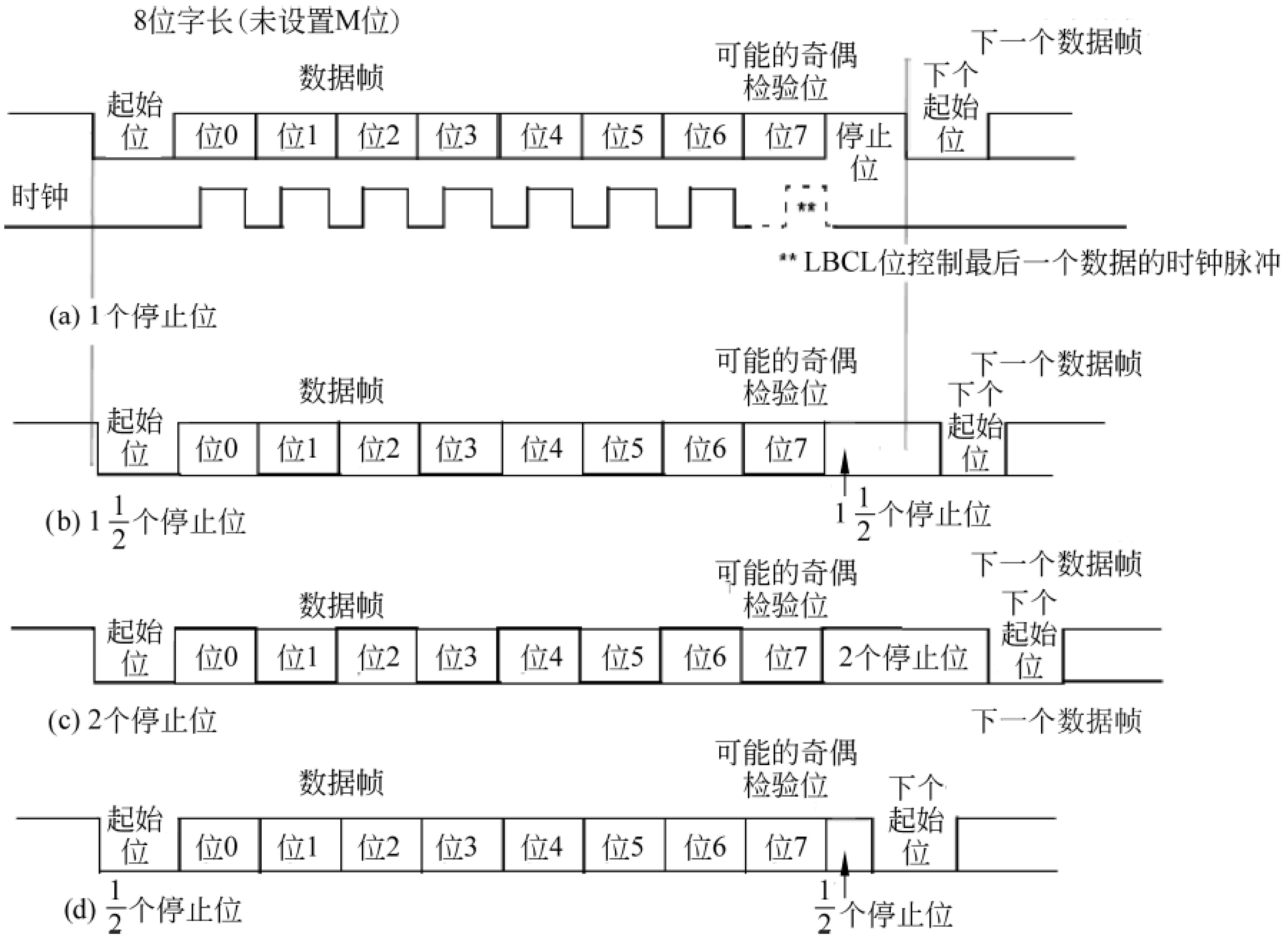


图 8-10　停止位的配置

- 1 个停止位：停止位位数的默认值。
- 2 个停止位：可用于常规 USART 模式、单线模式以及调制解调器模式。
- 0.5 个停止位：在智能卡模式下接收数据时使用。
- 1.5 个停止位：在智能卡模式下发送和接收数据时使用。

如图 8-5 所示，空闲帧包括了停止位，断开帧是 10 位低电平（M＝0）或 11 位低电平（M＝1），后跟停止位。

2. 单字节通信

发送一个字节时，需要将数据写入到数据寄存器 USART_DR，USART 控制器通过 TXE 和 TC 表示发送状态。TXE 是发送寄存器空指示标志，当发送寄存器 TDR 没有数据时置 1，清零 TXE 位是通过对数据寄存器的写操作来完成的。TXE 位由硬件来设置，它表明：

（1）数据已经从 TDR 移送到移位寄存器，数据发送已经开始；

（2）TDR 寄存器被清空；

(3) 下一个数据可以被写进 USART_DR 寄存器而不会覆盖先前的数据。

如果中断允许标志 TXEIE 位被设置为 1,则当 TXE 为 1 时将产生一个中断。如果此时 USART 正在发送数据,对 USART_DR 寄存器的写操作把数据存进 TDR 寄存器,并在当前传输结束时把该数据复制进移位寄存器。如果此时 USART 没有在发送数据,处于空闲状态,对 USART_DR 寄存器的写操作直接把数据放进移位寄存器,数据传输开始,TXE 位立即被置为 1。

当一个数据发送完成(停止位发送后)并且设置了 TXE 位为 1,TC 位被置为 1,TC 位用来表示数据传输已经完成,即当移位寄存器的最后一个数据发出以后,TC 被置 1,如果 USART_CR1 寄存器中的中断允许标志 TCIE 位被置 1 时,则会产生中断。在 USART_DR 寄存器中写入了最后一个数据字后,在关闭 USART 模块之前,必须先等待 TC=1。

发送时 TC/TXE 的状态变化如图 8-11 所示。

8.3.2 接收器

USART 可以根据 USART_CR1 寄存器的 M 位配置情况,接收 8 位或 9 位的数据字。

1. 起始位侦测

在 USART 中,如果辨认出一个特殊的采样序列,那么就认为侦测到一个起始位。该序列为 1110X0X0X0000,如图 8-12 所示。如果该序列不完整,那么接收端将退出起始位侦测并回到空闲状态(不设置标志位),等待判断下一个起始位。

在起始位的判断中,采用了 16 倍波特率时钟和过采样,如果 3 个采样点都为 0(在第 3、5、7 位的第一次采样,和在第 8、9、10 的第二次采样都为 0),则确认收到起始位,这时设置接收数据不为空标志位 RXNE=1,如果接收中断使能 RXNEIE=1,则产生中断。

如果两次 3 个采样点上仅有 2 个是 0 或一次 3 个采样点上仅有 2 个是 0(第 3、5、7 位的采样点和第 8、9、10 位的采样点),那么起始位仍然是有效的,但是会设置噪声标志位 NE。如果不能满足这个条件,则中止起始位的侦测过程,接收器会回到空闲状态(不设置标志位)。

2. 字符接收

在 USART 接收期间,数据的最低有效位首先从 RX 脚移进。USART_DR 寄存器的读操作实际上是对接收数据寄存器 RDR 的操作。接收过程中,USART 用接收数据寄存器非空标志 RXNE 表示数据是否收到,当一字符被接收到时:

- RXNE 位被置位。它表明移位寄存器的内容被转移到 RDR,即数据已经被接收并且可以被读出。
- 如果 RXNEIE 位被设置,产生中断。
- 在接收期间如果检测到各种错误,则相应的错位标志位被置 1。
- 在多缓冲器通信时(接收缓冲区是一个队列),RXNE 在每个字节接收后被置 1,并由 DMA 对数据寄存器的读操作而清零。
- 在单缓冲器模式里(接收缓冲区只能存储一个 8 位或 9 位的数据字),由软件读 USART_DR 寄存器完成对 RXNE 位清除,也可以通过对它写 0 来清除。RXNE 位必须在下一字符接收结束前被清零,以避免溢出错误。

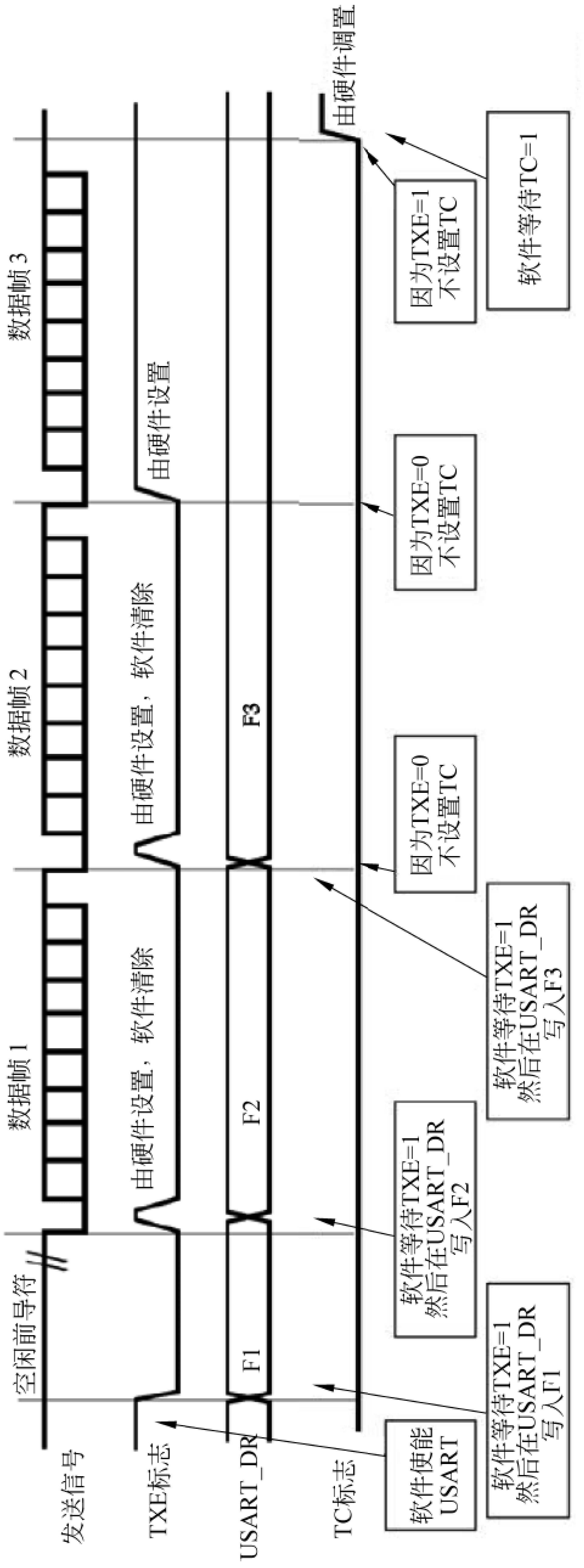


图 8-11 数据连续发送时 TXE 和 TC 的状态变化

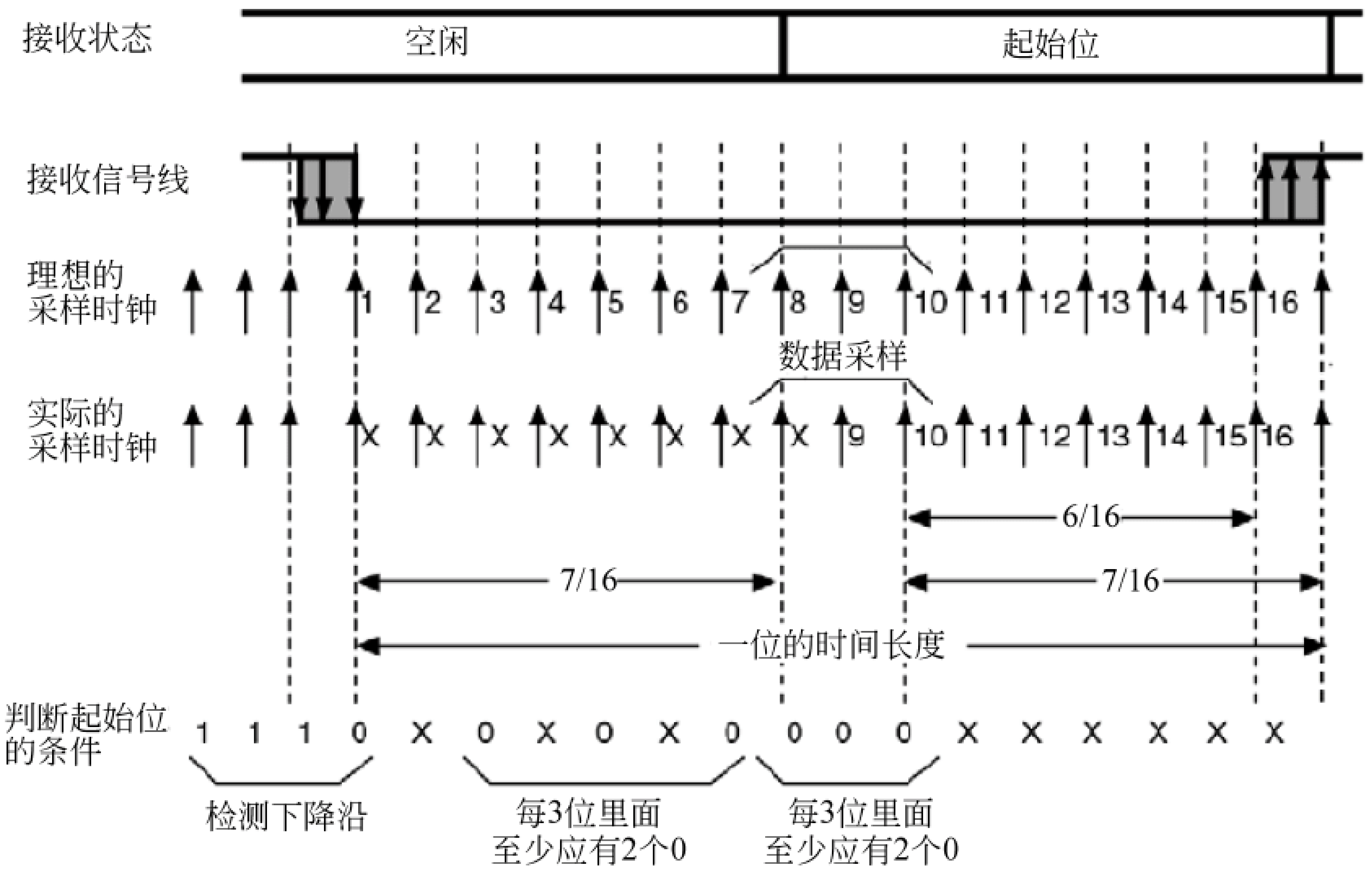


图 8-12 起始位检测

如果接收到一个断开符号，则 USART 以帧错误处理，如果收到空闲帧，其处理步骤和接收到普通数据帧一样，但如果空闲中断标志 IDLEIE 位被置 1 则将产生一个中断。

3. 接收采样时钟和过采样选择

STM32L152 USART 控制器支持 8 倍波特率和 16 倍波特率采样时钟，由控制寄存器 USART_CR1 的 OVER8 选择。异步传输的时钟精度虽然要求不高，但是也有一个容忍范围，采样时钟越高，容忍范围就越宽，但总线时钟最高 32MHz，因此数据传输的波特率必然受到限制。当 OVER8=0 时，采用 16 倍波特率采样时钟，最高波特率为 2Mb/s；当 OVER8=1 时，采用 8 倍波特率采样时钟，最高波特率可达 4Mb/s，但此时对时钟偏差的要求较高。

此外，为了提高数据抗干扰能力，可以通过控制寄存器 USART_CR3 的 ONEBIT 域开启三次采样功能。当 ONEBIT 置为 1 时，在每个有效数据位的最中间位置进行一次采样作为接收数据；当 ONEBIT 置为 0 时，在每个有效数据位的最中间进行三次采样，并对三次采样值进行投票判断，决定最终的有效数据位取值。如图 8-13 所示，16 倍波特率采样下，在时钟的 8、9、10 三个时刻进行 RX 数据采样，8 倍波特率时在时钟的 4、5、6 时刻采样。

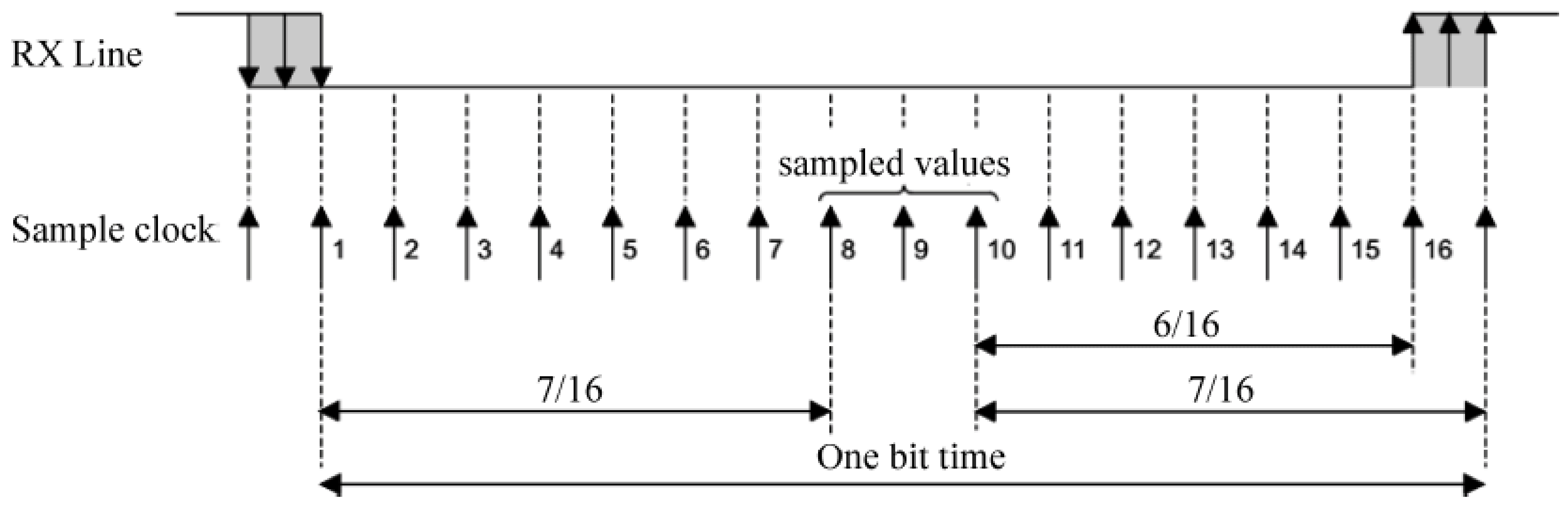


图 8-13 ONEBIT 三次采样时序

当三次采样的数据结果一致时(000 或 111),数据没有噪声,判定为有效;当三次采样的数据结果不一致时,按照少数服从多数的原则确定数据的有效电平,将数据认定为有噪声数据,并执行以下操作:

- 接收完数据,置 RXNE 位为 1,并同时设置噪声标志 NE 为 1。
- 无效数据从移位寄存器传送到 USART_DR 寄存器。
- 在单个字节通信情况下,NE 不会产生中断。然而,因为 NE 标志位和 RXNE 标志位是同时被设置的,RXNE 将产生中断。在多缓冲器通信情况下,如果已经设置了 USART_CR3 寄存器中 EIE 位,将产生一个中断。
- 先读状态寄存器 USART_SR,再读数据寄存器 USART_DR,将清除 NE 标志位。

图 8-14 是在不同采样频率和是否启用三次采样的情况下串口波特率能容忍的最大时钟误差,由图 8-14 可见,在 16 倍波特率相对 8 倍波特率时钟偏差的容忍度更大;三次采样比一次采样时钟偏差的容忍度更大。

M bit	OVER8 bit = 0		OVER8 bit = 1	
	ONEBIT=0	ONEBIT=1	ONEBIT=0	ONEBIT=1
0	3.75%	4.375%	2.50%	3.75%
1	3.41%	3.97%	2.27%	3.41%

图 8-14　串口能容忍的最大时钟误差

4. 接收中的错误

1) 溢出错误

如果 RXNE 还没有被复位,又接收到一个字符,则发生溢出错误。数据只有当 RXNE 位被清零后才能从移位寄存器转移到 RDR 寄存器。RXNE 标记是接收到每个字节后被置 1 的。如果下一个数据已被收到或先前 DMA 请求还没被服务时,RXNE 标志仍是置 1 的,溢出错误产生。当溢出错误产生时:

- 溢出错误标志位 ORE 被置 1;
- 接收数据寄存器 RDR 内容不会丢失,读 USART_DR 寄存器仍能得到先前的数据;
- 移位寄存器中以前的内容将被覆盖,随后接收到的数据都将丢失;
- 如果 RXNEIE 位被设置或 EIE 和 DMAR 位都被设置,中断产生;
- 顺序执行对 USART_SR 和 USART_DR 寄存器的读操作,可复位 ORE 位。

2) 噪音错误

当 ONEBIT 被置 1 启用三次采样时,若收到的三个采样值不一致,则产生噪声错误。噪声错误发生时,数据仍然被报错到 RDR 寄存器,但会置 NE 标志位,由用户决定是否使用该数据。

3) 帧错误

由于没有同步上或大量噪音的原因,停止位没有在预期的时间上接和收识别出来时产生帧错误,当帧错误被检测到时:

- FE 位被硬件置 1;

- 无效数据从移位寄存器传送到 USART_DR 寄存器，RXNE 置为 1；
- 在单字节通信时，FE 没有中断产生，但由于 RXNE 位置 1 产生中断，可通过 RXNE 中断服务对 FE 进行判断。在多缓冲器通信情况下，如果 USART_CR3 寄存器中 EIE 位被置位的话，将产生中断；
- 顺序执行对 USART_SR 和 USART_DR 寄存器的读操作，可复位 FE 位。

8.3.3 校验控制

设置 USART_CR1 寄存器上的 PCE 位，可以使能奇偶控制（发送时生成一个奇偶位，接收时进行奇偶校验）。根据 M 位定义的帧长度和是否启用校验，USART 的帧格式如表 8-2 所示。

表 8-2 启用/不启用校验时的数据帧格式

M 位	PCE 位	USART 帧
0	0	|起始位 | 8 位数据 | 停止位 |
0	1	|起始位 | 7 位数据 | 奇偶检验位 | 停止位 |
1	0	|起始位 | 9 位数据 | 停止位 |
1	1	|起始位 | 8 位数据 | 奇偶检验位 | 停止位 |

偶校验指的是校验位使得一帧中的 7 或 8 个数据位以及校验位中 1 的个数为偶数。奇校验指的是校验位使得一帧中的 7 或 8 个数据位以及校验位中 1 的个数为奇数。例如数据为 00110101，有 4 个 1，如果选择偶校验，校验位将是 0。如果选择奇校验，校验位将是 1。

如果启用校验，写进数据寄存器 TDR 的数据的最后一位被校验位替换后发送出去，如果奇偶校验失败，状态寄存器 USART_SR 寄存器中的 PE 标志被置 1，如果控制寄存器 USART_CR1 寄存器的 PEIE 被置 1，则产生一个中断。

8.3.4 硬件流控制

利用 nCTS 输入和 nRTS 输出可以控制两个设备间的串行数据流。通过将控制寄存器 UASRT_CR3 中的 RTSE 和 CTSE 置 1 可以分别独立地使能 RTS 和 CTS 流控制。

1. RTS 流控制

如图 8-15 所示，如果 RTS 流控制被使能，只要 USART 接收器准备好接收新的数据，nRTS 就变成有效（低电平）。当接收寄存器内有数据到达时，nRTS 被置无效（高电平），由此表明希望在当前帧结束时停止数据传输。

2. CTS 流控制

如图 8-16 所示，如果 CTS 流控制被使能，发送器在发送下一帧前检查 nCTS 输入。如果 nCTS 有效（低电平），则下一个数据被发送，否则下一帧数据不被发出去。若 nCTS 在传

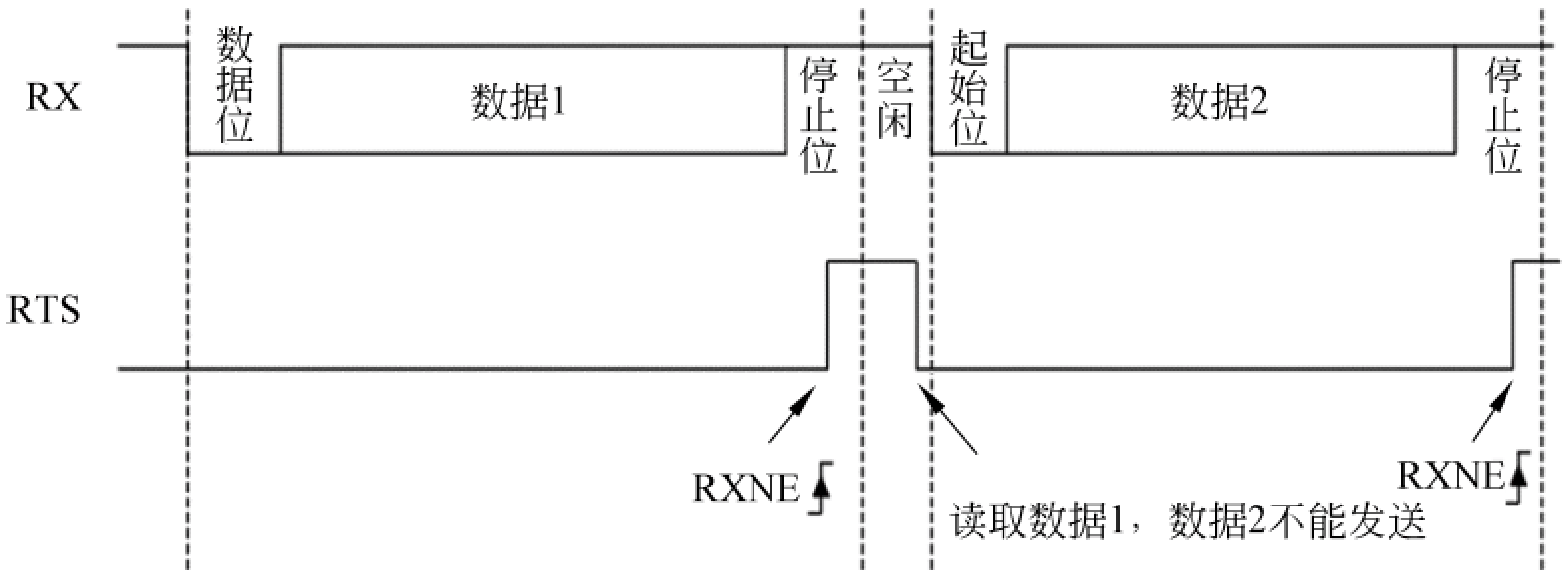


图 8-15　RTS 流控制时序

输期间被变成无效(高电平)，当前的传输完成后停止发送。在启用 CTS 流控制时，只要 nCTS 线的输入状态发生变化，硬件就自动设置 CTSIF 状态位为 1。如果设置了 USART_CT3 寄存器的 CTSIE 位，则产生中断。

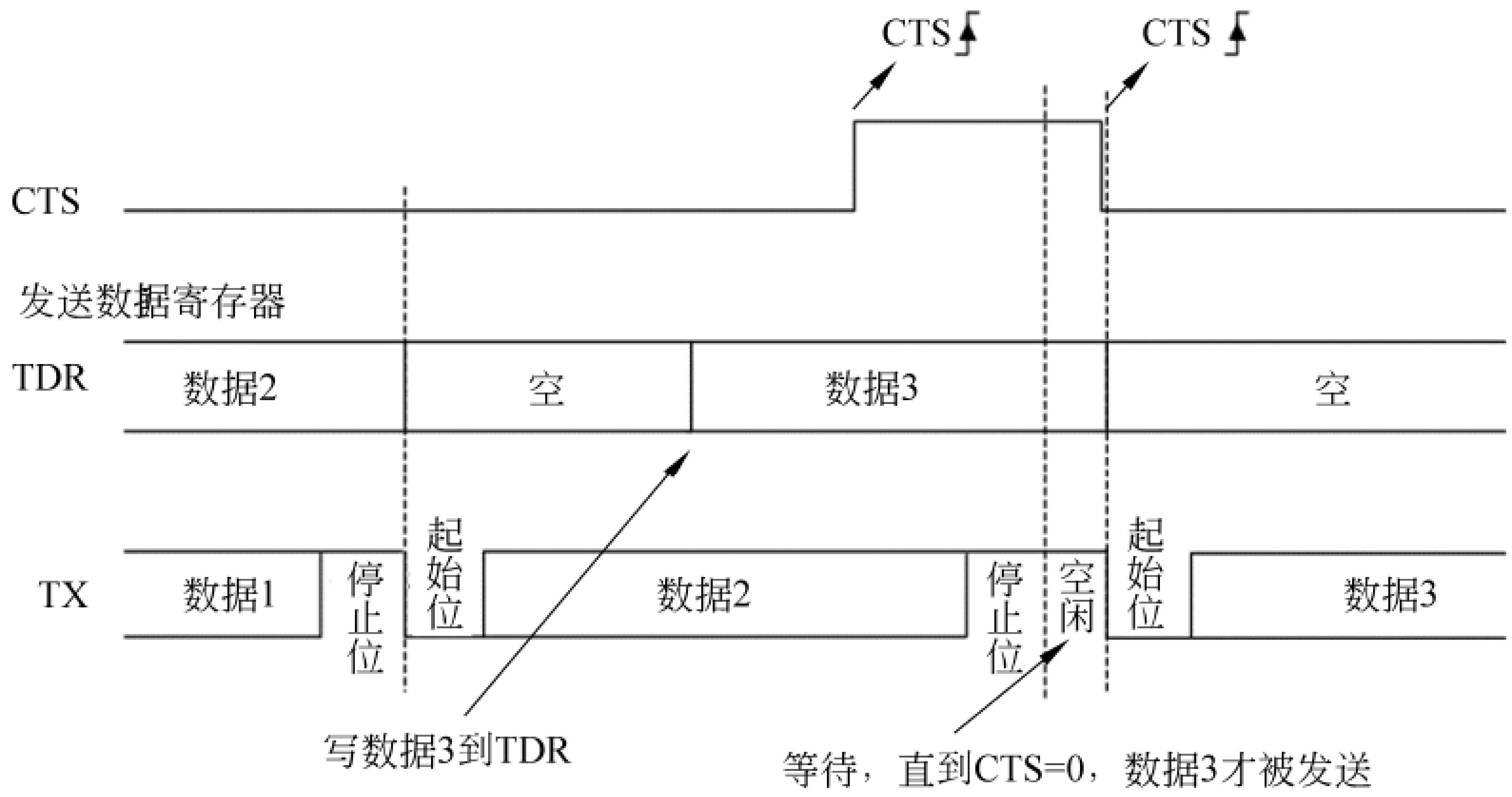


图 8-16　CTS 流控制时序

8.3.5　USART 中断请求

如表 8-3 所示，USART 控制器内部有多达 10 个中断事件，每个事件都有一个中断使能位用于决定是否可以向 NVIC 发起中断请求。各种中断事件被连接到同一个中断向量，其连接关系如图 8-17 所示。

表 8-3　USART 中断源

中断事件	事件标志	使能位
发送数据寄存器空	TXE	TXEIE
CTS 标志	CTS	CTSIE

续表

中断事件	事件标志	使能位
发送完成	TC	TCIE
接收数据就绪可读	RXNE	RXNEIE
检测到数据溢出	ORE	RXNEIE
检测到空闲线路	IDLE	IDLEIE
奇偶检验错	PE	PEIE
断开标志	LBD	LBDIE
DMA 多缓存区通信下的噪声标志、溢出错误和帧错误	NE 或 ORT 或 FE	EIE

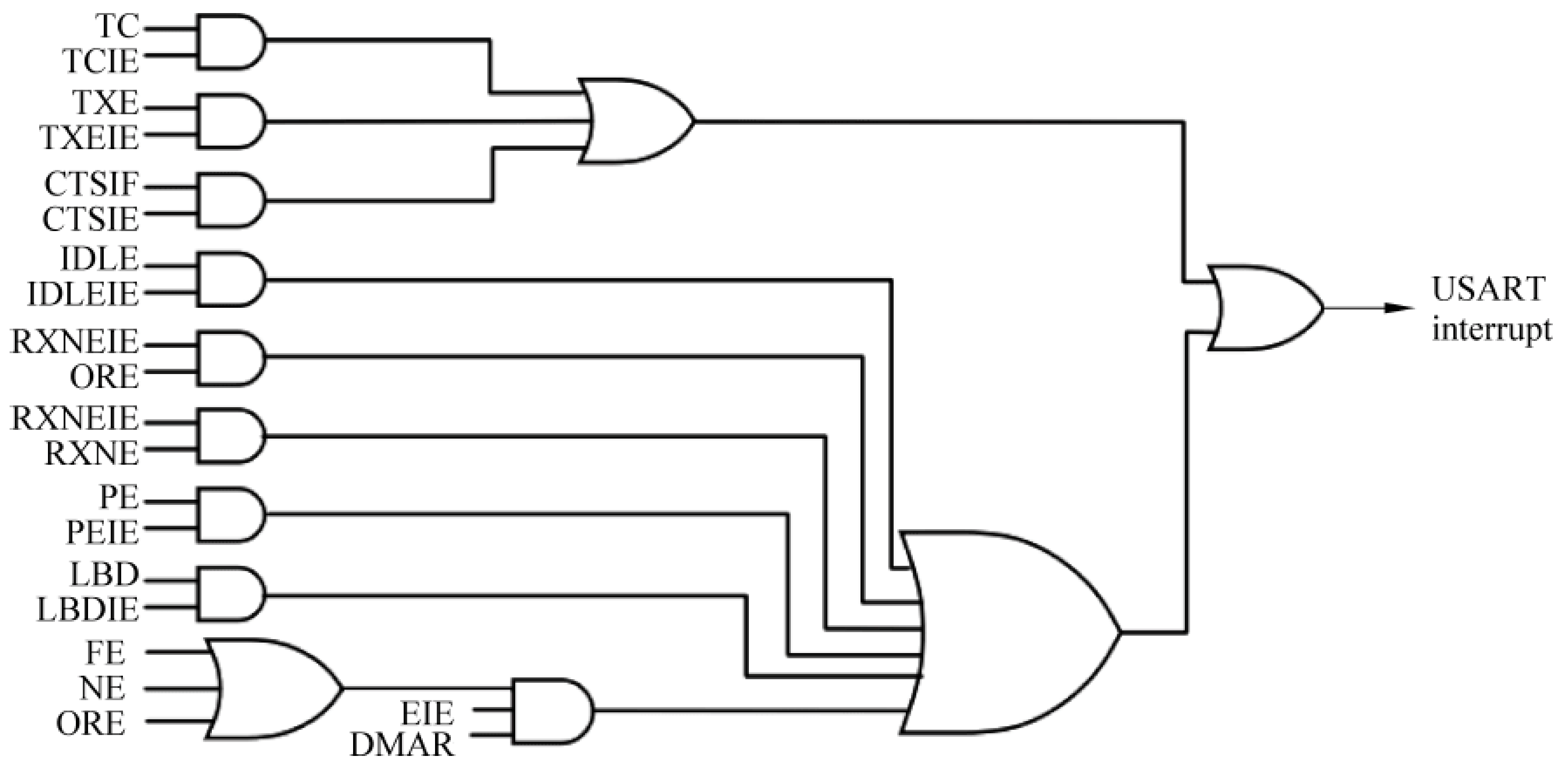


图 8-17　USART 内部中断源连接关系

8.4　USART 寄存器

USART 控制器的寄存器如表 8-4 所示。

表 8-4　USART 控制器寄存器

寄存器名称	地址偏移量	功　能	复位值
状态寄存器(USART_SR)	0x00	状态标志位	0x00C0 0000
数据寄存器(USART_DR)	0x04	发送和接收数据	0xXXXX XXXX
波特率寄存器(USART_BRR)	0x08	配置传输波特率因子	0x0000 0000
控制寄存器(USART_CR1)	0x0C	配置发送、接收及中断	0x0000 0000

续表

寄存器名称	地址偏移量	功　　能	复　位　值
控制寄存器(USART_CR2)	0x10	配置停止位、地址、同步时钟及LIN	0x0000 0000
控制寄存器(USART_CR3)	0x14	配置采样、流控制、红外和DMA	0x0000 0000
保护时间和分频寄存器(USART_GTPR)	0x18	配置智能卡和红外模式的保护时间和频率	0x0000 0000

1. 状态寄存器 USART_SR

状态寄存器 USART_SR 如图 8-18 所示,其有效域定义如下:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CTS	LBD	TXE	TC	RXNE	IDLE	ORE	NF	FE	PE
						rc_w0	rc_w0	r	rc_w0	rc_w0	r	r	r	r	r

图 8-18　状态寄存器

CTS:CTS 标志位,该位为 0 表示 nCTS 状态线上没有变化,为 1 表示 nCTS 状态线上发生变化。如果设置了 CTSE 位,当 nCTS 输入变化状态时,该位被硬件置 1。由软件将其清 0。如果 USART_CR3 中的 CTSIE 为 1,则产生中断。

LBD:LIN 断开检测(LIN Break Detect),当探测到 LIN 断开时,该位由硬件置 1,由软件清 0。如果 USART_CR3 中的 LBDIE=1,则产生中断。

TXE:发送寄存器空标志位,当 TDR 寄存器中的数据被硬件转移到移位寄存器的时候,该位被硬件置 1。如果 USART_CR1 寄存器中的 TXEIE 为 1,则产生中断。对 USART_DR 的写操作,将该位清 0。该位为 0 表示数据还没有被转移到移位寄存器,为 1 表示数据已经被转移到移位寄存器。

TC:发送完成标志位,该位为 0 表示发送还未完成,1 表示发送完成。当包含有数据的一帧发送完成后,并且 TXE=1 时,由硬件将该位置 1。如果 USART_CR1 中的 TCIE 为 1,则产生中断。由软件序列清除该位(先读 USART_SR,然后写 USART_DR)。TC 位也可以通过写入 0 来清除(多缓存通信中使用)。

RXNE:读数据寄存器非空标志位 RXNE,该位为 0 表示数据没有收到,1 表示收到数据,可以读出。当 RDR 移位寄存器中的数据被转移到 USART_DR 寄存器中,该位被硬件置位。如果 USART_CR1 寄存器中的 RXNEIE 为 1,则产生中断。对 USART_DR 的读操作可以将该位清零。RXNE 位也可以通过写入 0 来清除(在多缓存通信中使用)。

IDLE:总线空闲标志位,该位为 0 表示没有检测到空闲总线,为 1 表示检测到空闲总线。当检测到总线空闲时,该位被硬件置 1。如果 USART_CR1 中的 IDLEIE 为 1,则产生中断。由软件序列清除该位(先读 USART_SR,然后读 USART_DR)。

ORE:过载错误标志位,该位为 0 表示没有过载错误,为 1 表示检测到过载错误。当 RXNE 仍然是 1 的时候,当前被接收在移位寄存器中的数据,需要传送至 RDR 寄存器时,

硬件将该位置1。如果USART_CR1中的RXNEIE为1,则产生中断。由软件序列将其清零(先读USART_SR,然后读USART_CR)。该位被置位时,RDR寄存器中的值不会丢失,但是移位寄存器中的数据会被覆盖。如果设置了EIE位,在多缓冲器通信模式下,ORE标志置位会产生中断。

NE:噪声错误标志,该位为0表示没有检测到噪声,为1表示检测到噪声。在接收到的帧检测到噪音时,由硬件对该位置1。由软件序列对其清0(先读USART_SR,再读USART_DR)。该位不会产生中断,但因为它和RXNE一起出现,硬件会在设置RXNE标志时产生中断。在多缓冲区通信模式下,如果设置了EIE位,则设置NE标志时会产生中断。

FE:帧错误标志位FE,该位为0表示没有检测到帧错误,为1表示检测到帧错误或者break符。当检测到同步错位、过多的噪声或者检测到断开符,该位被硬件置1。由软件序列将其清零(先读USART_SR,再读USART_DR)。该位不会产生中断,但因为它和RXNE一起出现,硬件会在设置RXNE标志时产生中断。如果当前传输的数据既产生了帧错误,又产生了过载错误,硬件还是会继续该数据的传输,并且只设置ORE标志位。在多缓冲区通信模式下,如果设置了EIE位,则设置FE标志时会产生中断。

PE:校验错误标志位,该位为0表示没有奇偶校验错误,为1表示检测到奇偶校验错误。在接收模式下,如果出现奇偶校验错误,硬件对该位置1。由软件序列对其清0(依次读USART_SR和USART_DR)。如果USART_CR1中的PEIE为1,则产生中断。

2. 数据寄存器 USART_DR

数据寄存器如图8-19所示,由两个寄存器组成的,一个给发送用(TDR),一个给接收用(RDR),该寄存器兼具读和写的功能。TDR寄存器提供了内部总线和输出移位寄存器之间的并行接口,RDR寄存器提供了输入移位寄存器和内部总线之间的并行接口。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留							DR[8:0]								
							rw	rw	rw	rw	rw	rw	rw	rw	rw

图8-19 数据寄存器

数据寄存器的有效域为DR[8:0],表示数据值,包含了发送或接收的数据。当使能校验位(USART_CR1中PCE位被置位)进行发送时,写到MSB的值(根据数据的长度不同,MSB是第7位或者第8位)会被后来的校验位取代。当使能校验位进行接收时,读到的MSB位是接收到的校验位。

3. 波特率配置寄存器 USART_BRR

波特率配置寄存器USART_BRR如图8-20所示,其有效域定义如下:

DIV_Mantissa[11:0]:USARTDIV的整数部分,定义了USART分频器除法因子(USARTDIV)的整数部分。

DIV_Fraction[3:0]:USARTDIV的小数部分,定义了USART分频器除法因子(USARTDIV)的小数部分。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIV_Mantissa[11:0]												DIV_Fraction[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 8-20　波特率寄存器

4. 控制寄存器 USART_CR1

控制寄存器 USART_CR1 如图 8-21 所示，其有效域定义如下：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OVER8	Reserved	UE	M	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	RWU	SBK
rw	Res.	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 8-21　控制寄存器 1

OVER8：采样时钟选择位，该位为 1 表示采用发送/接收时钟的 8 分频作为波特率，为 0 表示采用发送/接收时钟的 16 分频作为波特率。

UE：USART 使能（USART enable），该位被置 0，表示禁用 USART 模块，在当前字节传输完成后 USART 的分频器和输出停止工作。置 1 表示 USART 模块使能。

M：字长，该位定义了数据字的长度，由软件对其置 1 和清 0，0 表示 8 个数据位，1 表示 9 个数据位，在数据传输过程中(发送或者接收时)，不能修改这个位。

WAKE：静默模式唤醒方法，这位决定将 USART 从静默模式唤醒的方法，0 表示被空闲总线唤醒，1 表示被地址标记唤醒，由软件对该位置 1 和清 0。

PCE：检验控制使能（Parity control enable），0 表示禁止校验控制，1 表示使能校验控制。用该位选择是否进行硬件校验控制(对于发送来说就是校验位的产生；对于接收来说就是校验位的检测)。当使能了该位，在发送数据的最高位(如果 M=1，最高位就是第 9 位；如果 M=0，最高位就是第 8 位)插入校验位；对接收到的数据检查其校验位。一旦设置了该位，当前字节传输完成后，校验控制才生效。

PS：校验选择（Parity selection），当校验控制 PCE 置 1 后，PS 用于选择采用偶校验还是奇校验。0 表示偶校验，1 表示奇校验。

PEIE：PE 中断使能（PE interrupt enable），0 表示禁止产生中断，1 表示当 USART_SR 中的 PE 为 1 时，产生 USART 中断。

TXEIE：发送缓冲区空中断使能（TXE interrupt enable），0 表示禁止产生中断，1 表示当 USART_SR 中的 TXE 为 1 时，产生 USART 中断。

TCIE：发送完成中断使能（Transmission complete interrupt enable），0 表示禁止产生中断，1 表示当 USART_SR 中的 TC 为 1 时，产生 USART 中断。

RXNEIE：接收缓冲区非空中断使能（RXNE interrupt enable），0 表示禁止产生中断，1 表示当 USART_SR 中的 ORE 或者 RXNE 为 1 时，产生 USART 中断。

IDLEIE：IDLE 中断使能（IDLE interrupt enable），0 表示禁止产生中断，1 表示当 USART_SR 中的 IDLE 为 1 时，产生 USART 中断。

TE：发送使能（Transmitter enable），0 表示禁止发送，1 表示使能发送，当 TE 被设置后，在真正发送开始之前，有一个比特时间的延迟。

RE：接收使能（Receiver enable），0 表示禁止接收，1 表示使能接收，并开始搜寻 RX 引脚上的起始位。

RWU：接收唤醒（Receiver wakeup），该位用来决定是否把 USART 置于静默模式，0 表示处于正常工作模式，1 表示接收器处于静默模式，当唤醒序列到来时，硬件将其清零。

SBK：发送断开帧（Send break），使用该位来发送断开字符。0 表示没有发送断开字符，1 表示将要发送断开字符。

5. 控制寄存器 USART_CR2

控制寄存器 USART_CR2 如图 8-22 所示，其有效域定义如下：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Res.	LINEN	STOP[1:0]		CLKEN	CPOL	CPHA	LBCL	Res.	LBDIE	LBDL	Res.	ADD[3:0]			
	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw

图 8-22　控制寄存器 2

LINEN：LIN 模式使能（LIN mode enable），0 表示禁止 LIN 模式，1 表示开启。

STOP：停止位（STOP bits），这 2 位用来设置停止位的位数，00 表示 1 个停止位，01 表示 0.5 个停止位，10 表示 2 个停止位，11 表示 1.5 个停止位，同步和异步串行模式智能使用 1 个或 2 个停止位。

CLKEN：时钟使能（Clock enable），该位用来使能 CK 引脚，0 表示禁止 CK 引脚，1 表示使能 CK 引脚。UART4 和 UART5 上不存在这一位。

CPOL：时钟极性（Clock polarity），在同步模式下，可以用该位选择 SLCK 引脚上时钟输出的极性。和 CPHA 位一起配合来产生需要的时钟/数据的采样关系，0 表示总线空闲时 CK 引脚上保持低电平，1 表示总线空闲时 CK 引脚上保持高电平。

CPHA：时钟相位（Clock phase），在同步模式下，可以用该位选择 SLCK 引脚上时钟输出的相位。0 表示在时钟的第一个边沿进行数据捕获，1 表示在时钟的第二个边沿进行数据捕获。

LBCL：最后一个字节的时钟控制，在同步模式下，使用该位来控制是否在 CK 引脚上输出最后发送的那个数据字节（MSB）对应的时钟脉冲，0 表示不输出，1 表示输出。UART4 和 UART5 没有 CPOL、CPHA 和 LBCL 域。

LBDIE：LIN 断开符检测中断使能（LIN break detection interrupt enable），断开符中断屏蔽（使用断开分隔符来检测断开符），0 表示禁止中断，1 表示 USART_SR 寄存器中的 LBD 为 1 就产生中断。

LBDL：LIN 断开符检测长度（LIN break detection length），该位用来选择是 11 位还

是 10 位的断开符检测，0 表示 10 位，1 表示 11 位。

ADD[3：0]：本设备的 USART 节点地址，在多处理器通信下的静默模式中使用的，使用地址标记来唤醒某个 USART 设备。

6. 控制寄存器 USART_CR3

控制寄存器 USART_CR3 如图 8-23 所示，其有效域定义如下：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				ONEBIT	CTSIE	CTSE	RTSE	DMAT	DMAR	SCEN	NACK	HDSEL	IRLP	IREN	EIE
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 8-23　控制寄存器 3

ONEBIT：过采样使能位，为 1 表示只进行一次采样，为 0 表示进行三次采样。

CTSIE：CTS 中断使能（CTS interrupt enable），0 表示禁止中断，1 表示 USART_SR 寄存器中的 CTS 为 1 时产生中断。UART4 和 UART5 上不存在这一位。

CTSE：CTS 使能（CTS enable），0 表示禁止 CTS 硬件流控制，1 表示 CTS 模式使能，只有 nCTS 输入信号有效(拉成低电平)时才能发送数据。如果在数据传输的过程中，nCTS 信号变成无效，那么发完这个数据后，传输就停止下来。如果当 nCTS 为无效时，往数据寄存器里写数据，则要等到 nCTS 有效时才会发送这个数据。UART4 和 UART5 上不存在这一位。

RTSE：RTS 使能（RTS enable），0 表示禁止 RTS 硬件流控制，1 表示 RTS 使能，只有接收缓冲区内有空余的空间时才请求下一个数据。当前数据发送完成后，发送操作就需要暂停下来。如果可以接收数据了，将 nRTS 输出置为有效（拉至低电平）。UART4 和 UART5 上不存在这一位。

DMAT：发送 DMA 使能(DMA enable Trasmit)，0 表示禁止发送时的 DMA，1 表示使能发送时的 DMA；UART4 和 UART5 上不存在这一位。

DMAR：DMA 使能接收（DMA enable receiver），0 表示禁止接收时 DMA，1 表示使能接收时的 DMA；UART4 和 UART5 上不存在这一位。

SCEN：智能卡模式使能（Smartcard mode enable），该位用来使能智能卡模式，0 表示禁止，1 表示使能。

NACK：智能卡 NACK 使能（Smartcard NACK enable），0 表示校验错误时，不发送 NACK，1 表示校验错误出现时发送 NACK。

HDSEL：半双工选择（Half-duplex selection），选择单线半双工模式，1 表示选择半双工模式。

IRLP：红外低功耗（IrDA low-power），该位用来选择普通模式还是低功耗红外模式，1 表示低功耗模式。

IREN：红外模式使能（IrDA mode enable），1 表示使能红外模式。

EIE：错误中断使能（Error interrupt enable），在多缓冲区通信模式下，当有帧错误、过载或者噪声错误时(USART_SR 中的 FE＝1，或者 ORE＝1，或者 NE＝1)产生中断的使能

位。0 表示禁止中断，1 表示只要 USART_CR3 中的 DMAR=1，并且 USART_SR 中的 FE=1，或者 ORE=1，或者 NE=1，则产生中断。

8.5 USART 数据传输配置

8.5.1 波特率计算

发送和接收方的波特率必须要配置为一致的数值，常用的波特率位 1200、2400、4800、9600、19200、57600、115200 等，其计算公式为：

$$\text{TX/RX baud}=\frac{f_{CK}}{8\times(2-\text{OVER8})\times\text{USARTDIV}}$$

其中 f_{CK} 为总线时钟，OVER8 为 16 倍波特率/8 倍波特率选择，USARTDIV 为配置给波特率寄存器 USART_BRR 的值，由 12 位整数和 4 位小数表示。

OVER8=0 时，USARTDIV = DIV_Mantissa[11：0] + DIV_Fraction[3：0]/16；

OVER8=1 时，USARTDIV = DIV_Mantissa[11：0] + DIV_Fraction[2：0]/8。

例如，f_{CK}=32MHz，采样时钟为 8 倍波特率，要配置波特率为 57600，则 USART_BRR 寄存器的值计算过程为：

$$\text{USARTDIV}=f_{CK}/\text{baud}/8 = 32\text{M}/57600/8 = 69.44$$

因此，DIV_Mantissa = 69，DIV_Fraction=0.44×8 = 3.52，取值为 4。实际波特率与所需要的波特率存在偏差。波特率在 16MHz 和 32MHz 下的配置值如图 8-24 和图 8-25 所示。

16倍采样时钟（OVER8=0）							
波特率		fPCLK=16MHz			fPCLK=32MHz		
序号	需求波特率/kb/s	实际波特率/kb/s	波特率寄存器值	误差%	实际波特率/kb/s	波特率寄存器值	误差/%
1	1.2	1.2	833.3125	0	1.2	1666.6875	0
2	2.4	2.4	416.6875	0	2.4	833.3125	0
3	9.6	9.598	104.1875	0.02	9.601	208.3125	0.01
4	19.2	19.208	52.0625	0.04	19.196	104.1875	0.02
5	38.4	38.369	26.0625	0.08	38.415	52.0625	0.04
6	57.6	57.554	17.375	0.08	57.554	34.75	0.08
7	115.2	115.108	8.6875	0.08	115.108	17.375	0.08
8	230.4	231.884	4.3125	0.64	230.216	8.6875	0.08
9	460.8	457.143	2.1875	0.79	463.768	4.3125	0.64
10	921.6	941.176	1.0625	2.12	914.286	2.1875	0.79
11	2	NA	NA	NA	2000	1	0
12	4	NA	NA	NA	NA	NA	NA

图 8-24　16 倍波特率下的 USARTDIV 取值

8倍采样时钟（OVER8=1）							
波特率		fPCLK=16MHz			fPCLK=32MHz		
序号	需求波特率/kb/s	实际波特率/kb/s	波特率寄存器值	误差/%	实际波特率/kb/s	波特率寄存器值	误差/%
1	1.2	1.2	1666.625	0	1.2	3333.375	0
2	2.4	2.4	833.375	0	2.4	1666.625	0
3	9.6	9.598	208.375	0.02	9.601	416.625	0.01
4	19.2	19.208	104.125	0.04	19.196	208.375	0.02
5	38.4	38.369	52.125	0.08	38.415	104.125	0.04
6	57.6	57.554	34.75	0.08	57.554	69.5	0.08
7	115.2	115.108	17.375	0.08	115.108	34.75	0.08
8	230.4	231.884	8.625	0.64	230.216	17.375	0.08
9	460.8	457.143	4.375	0.79	463.768	8.625	0.64
10	921.6	941.176	2.125	2.12	914.286	4.375	0.79
11	2	2000	1	0	2000	2	0
12	4	NA	NA	NA	4000	1	0

图 8-25　8 倍波特率下的 USARTDIV 取值

8.5.2　异步双向通信模式配置

1）发送配置

- 通过在 USART_CR1 寄存器上置位 UE 位来激活 USART；
- 编程 USART_CR1 的 M 位来定义字长；
- 在 USART_CR2 中编程停止位 STOP 的位数；
- 利用 USART_BRR 寄存器选择要求的波特率；
- 如果采用多缓冲器通信，配置 USART_CR3 中的 DMA 使能位(DMAT)，按多缓冲器通信中的描述配置 DMA 寄存器；
- 设置 USART_CR1 中的 TE 位，发送一个空闲帧作为第一次数据发送；
- 把要发送的数据写进 USART_DR 寄存器(此动作清除 TXE 位)，重复此步骤发送其他数据；
- 在 USART_DR 寄存器中写入最后一个数据字后，要等待 TC=1，它表示最后一个数据帧的传输结束。当需要关闭 USART 或需要进入停机模式之前，需要确认传输结束，避免破坏最后一次传输。

2）接收配置

- 将 USART_CR1 寄存器的 UE 置 1 来激活 USART；

- 编程 USART_CR1 的 M 位定义字长；
- 在 USART_CR2 中编写停止位 STOP 的个数；
- 利用波特率寄存器 USART_BRR 选择希望的波特率；
- 如果需多缓冲器通信，选择 USART_CR3 中的 DMA 使能位(DMAR)。按多缓冲器通信所要求的配置 DMA 寄存器；
- 设置 USART_CR1 的 RE 位。激活接收器，使它开始寻找起始位。

8.6 USART 帧传输协议

串口发送数据时是面向字节的，接收方接收数据时也是面向字节的，实际应用中，收发数据往往都是面向帧(若干字节)的。若一个嵌入式微控制器通过串口往 PC 发送一帧 100 字节数据，由于数据的异步性，字节之间的间隔无法保证，PC 在调用系统 API 来接收数据时，往往不能一次性接收完 100 字节，可能第一次接收 5 字节，第二接收 10 字节，虽然最后都能收到 100 字节。但无法从时间上判断是否是一个完整的数据；如果微控制器发送两帧数据，并且这两帧数据之间的时间间隔非常短，PC 有可能无法区别出两帧数据的边界了，因此我们需要在链路层进行帧格式设计。

典型的串行链路帧协议有点到点传输协议 PPP(Point to Point Protocol)、高级链路控制协议 HDLC(High Data Link Control Protocol)以及面向工业应用的 MODBUS 协议等。

8.6.1 串行链路帧格式设计

为解决异步串行数发送时无法判断多字节流组成的数据帧的头和尾的问题，我们引入两个特殊字符，即帧起始字符 SOF、帧结束字符 EOF，在发送数据时，将多字节数据打包，头部加一个 SOF 字符，尾部加一个 EOF 字符，如图 8-26 所示，这样，接收端检测数据帧的 SOF 字符，一旦检测到 SOF，则认为一个新数据帧的开始，直到接收到 EOF 字符为止。

图 8-26 链路帧格式设计

在串行传输中，有两种传输格式，一种是字符传输，另一种是二进制传输；当采用字符传输时，我们可以定义两个不出现在数据中的特殊字符，如不可见字符制表符、分隔符等，这样可以解决上述帧识别的问题，但在二进制传输中，数据部分可能的取值范围为 0x00～0xFF，因此数据中可能会出现 SOF 或 EOF，这样会出现帧识别错误，如图 8-27 所示。

为解决二进制传输中的问题，我们引入一个转义字符，在发送端的数据中，如果出现了

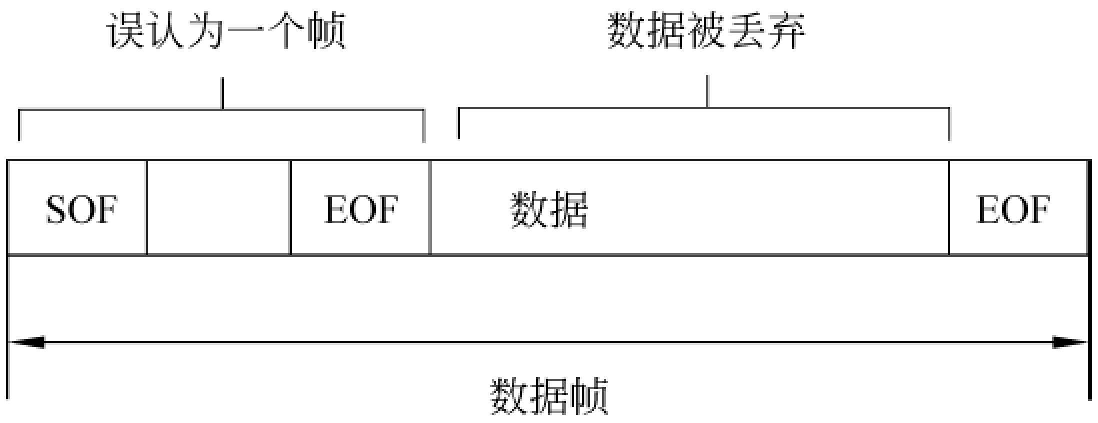


图 8-27　数据中出现 EOF 或 SOF 时帧识别错误

SOF 或 EOF，则插入一个转义字符 ESC 进行变换，若数据中出现了转义字符 ESC，则 ESC 也要进行转义，这样保证发送出去的数据帧中只有一个 SOF 和一个 EOF，接收方接收到数据后进行反转义，恢复数据，转义后的数据格式如图 8-28 所示。

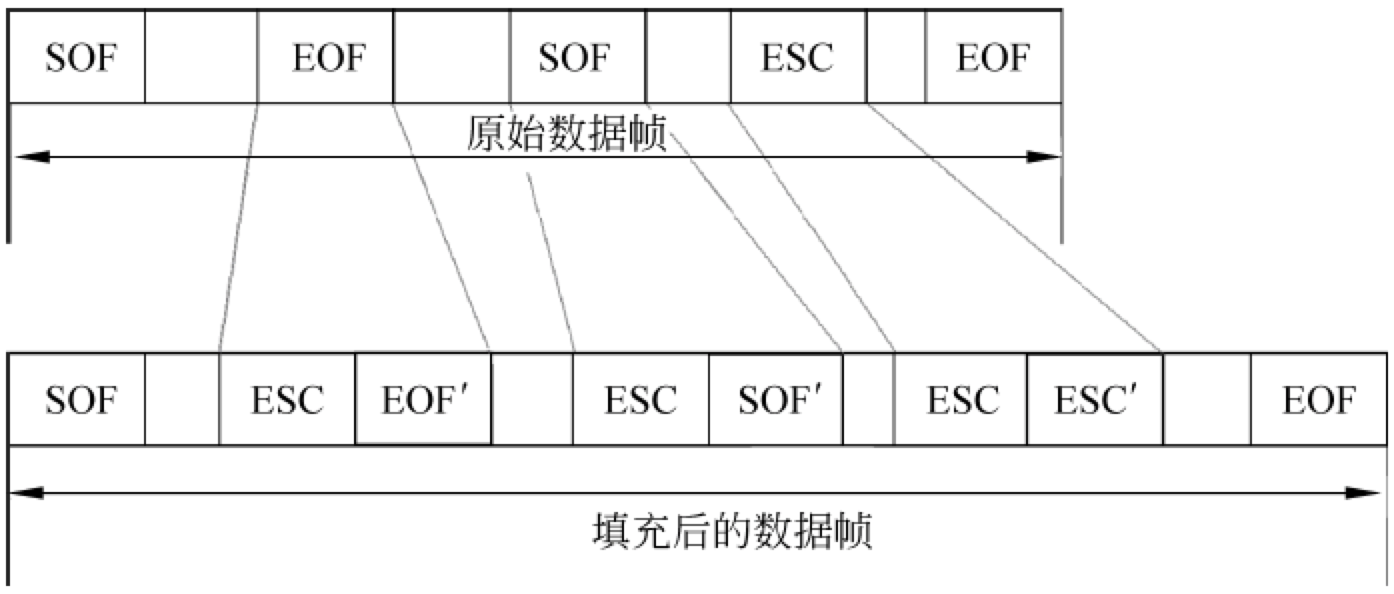


图 8-28　数据转以后的帧格式

1. PPP 协议的字符填充

PPP 协议中，SOF 和 EOF 均取值为 0x7e，ESC 取值为 0x7D，其转义规则如下：

- 如果数据中出现 0x7E，则插入一个 0x7D，将 0x7E 转变为 0x7D 0x7E^0x20 两个字符，即 0x7D 0x5E；
- 如果数据中出现 0x7D，则插入一个 0x7D，将 0x7D 转变为 0x7D 0x7D^0x20 两个字符，即 0x7D 0x5D；
- 如果数据中出现了小于 0x20 的数据，则将该数据转变成 0x7D 原始数据^0x20 两个字符。

^表示异或操作。一个典型例子如图 8-29 所示。

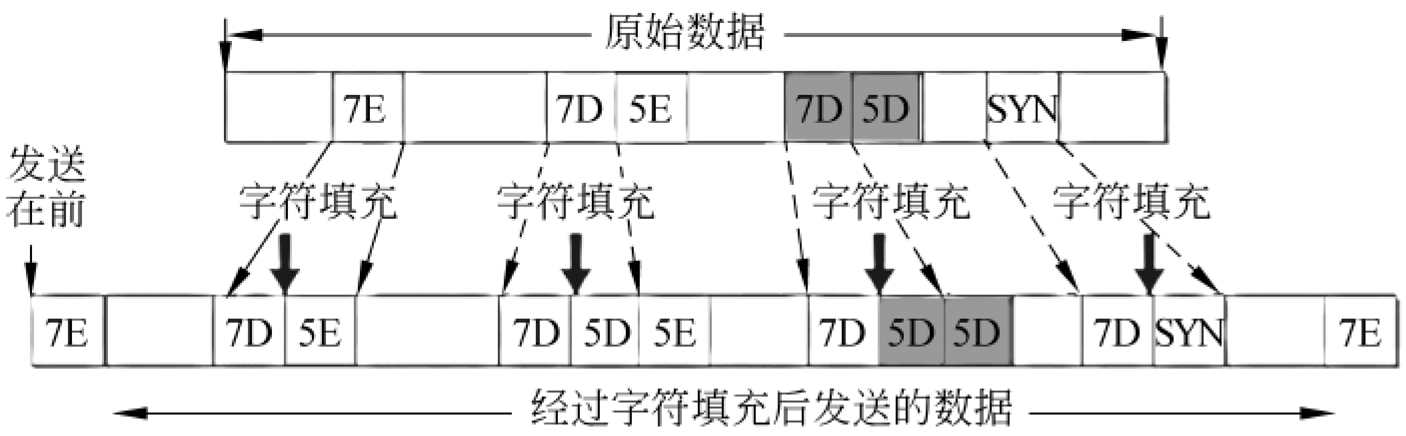


图 8-29　PPP 转义后的数据格式

PPP 数据的编码和解码算法如下：

```
#define PPP_FRAME_FLAG        (0x7E)                    /* 标识字符 */
#define PPP_FRAME_ESC         (0x7D)                    /* 转义字符 */
#define PPP_FRAME_ENC         (0x20)                    /* 编码字符 */
int ppp_encode(unsigned char *in, int in_len, unsigned char *out, int *out_len)
{
    unsigned char *pi, *po;
    int i, tmp_len;
    pi = in;
    po = out;
    tmp_len = in_len;
    for(i = 0; i < in_len; i++)
    {
        if( *pi == PPP_FRAME_FLAG || *pi == PPP_FRAME_ESC || *pi < 0x20 )
        {
            *po = PPP_FRAME_ESC;
            po++;
            tmp_len++;
            *po = *pi ^ PPP_FRAME_ENC;
        }
        else
            *po = *pi;
        pi++;
        po++;
    }
    *out_len = tmp_len;
    return 0;
}
int ppp_decode(unsigned char *in, int in_len, unsigned char *out, int *out_len)
{
    unsigned char *pi, *po;
    int i, tmp_len;
    pi = in;
    po = out;
    tmp_len = in_len;
    for(i = 0; i < in_len; i++)
    {
        if(*pi == PPP_FRAME_ESC)
        {
            pi++;
            tmp_len--;
            *po = *pi ^ PPP_FRAME_ENC;
```

```
            i++;
        }
        else
            *po = *pi;
        pi++;
        po++;
    }
    *out_len = tmp_len;
    return 0;
}
```

HDLC 协议的转义规则和 PPP 类似，只不过只进行 0x7E 和 0x7D 的转义，不对小于 0x20 的数据转义。除了底层转义外，HDLC 和 PPP 有帧格式定义，如图 8-30 所示。

HDLC

Flag	Address	Control	Data	FCS	Flag
1byte	1Byte	1/2bytes	1500bytes	2/4bytes	1byte

PPP

Flag	Address	Control	Protocol	Data	FCS	Flag
1byte	1Byte	1bytes	1/2bytes	1500bytes	2/4bytes	1byte

图 8-30　HDLC 和 PPP 帧格式定义

Flag 字段即为 SOF 和 EOF 字符，地址字段用来对通信设备进行寻址，Control 字段用于表示控制信息类型，PPP 中的 Protocol 字段用于表示 PPP 报文中封装的 payload(data 字段)的类型，最后的 FCS 字段为校验字节，用于对数据帧进行检错。

2. PPP/HDLC 的串口通信状态机

我们通常利用状态机来实现串行链路帧的发送和接收，图 8-31 是 HDLC 链路帧格式的发送和接收状态机，发送时，初始状态为空闲状态，首先发送 0x7E，进入帧头发送状态，发

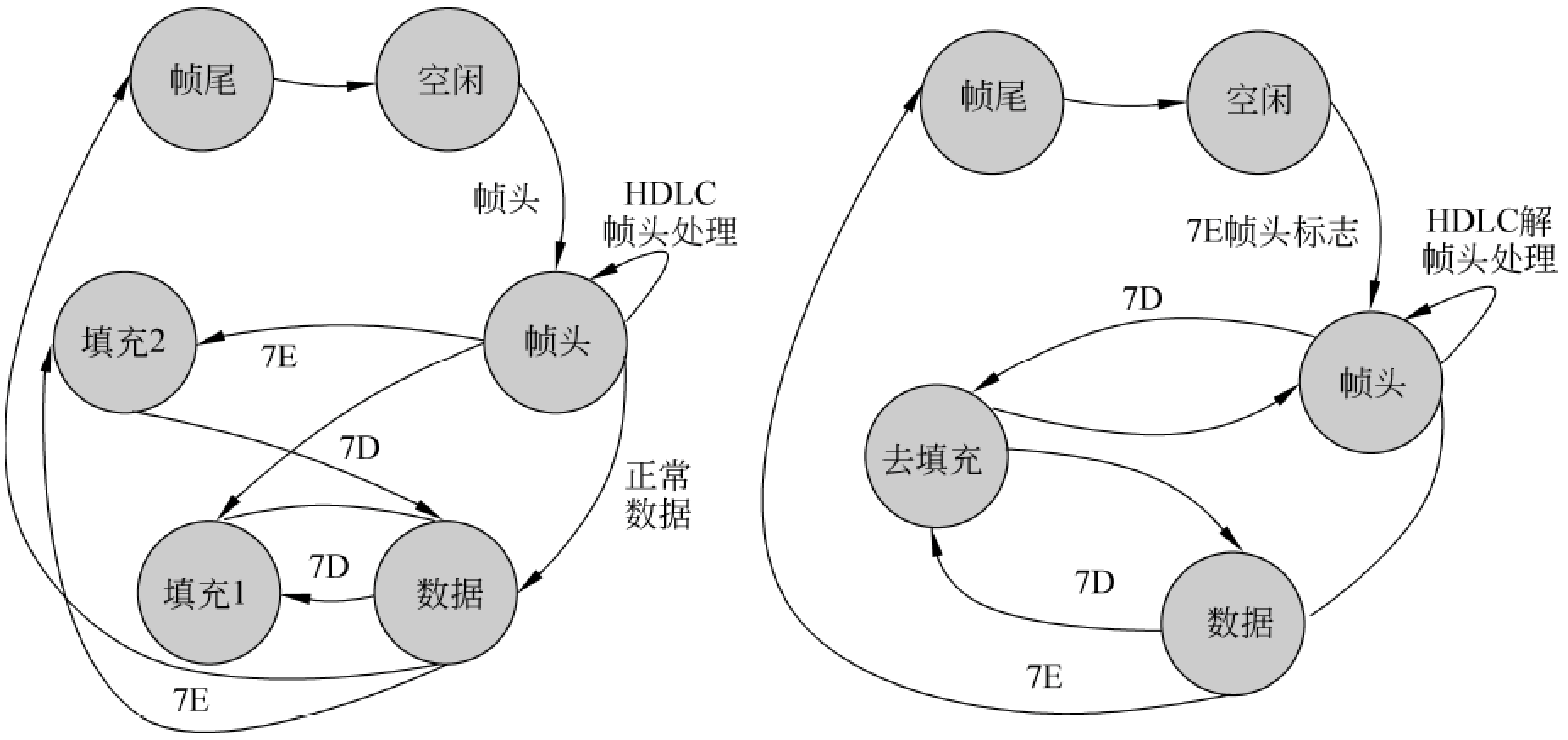


图 8-31　HDLC 串行发送和接收状态机

送完 Address、Control 后，进入数据发送状态(包括了 FCS)，在帧头和数据发送过程中，若数据中出现了 0x7D 或 0x7E，则进入转义状态进行转义，发送完所有数据后，发送 0x7E 帧尾结束。

接收时，状态机处于空闲状态，检测到 0x7E，则进入帧头处理状态，紧接着进入数据状态，若数据和帧头中出现了 0x7D 则进入转义状态，接收到 0x7E 后，一个帧接收结束。

8.6.2 MODBUS 帧格式

MODBUS 是一种主从模式的工业现场总线协议，允许一个主机最多连接 247 个从属控制器，支持 RS-485、RS-232、RS-422 和以太网物理层接口，通常用于 PLC，DCS 以及智能仪表的现场总线连接。

MODBUS 的支持 ASCII、RTU 和 TCP 模式，通常使用 ASCII 码模式，每一个数据为一个 ASCII 字符，使用 7b 表示字符，加上 1 个奇偶校验位，串口需配置成 8b 模式，并对串行数据流进行 LRC 校验。另外一种常用的是 RTU 模式，即传输的是二进制数据，采用 CRC 校验，优点是相同传输波特率下，比 ASCII 模式传输数据密度高，速率快，但实现和控制相对较为复杂。

1. ASCII 模式

ASCII 模式下和 HDLC 类似，定义了一个起始字符‘：’和两个结束字符回车 CR、换行 LF 表示数据帧的开始和结尾，所传送数据都是 ASCII 字符，用十六进制表示，即所传输的数据是由 0123456789ABCDEF 等 16 个 ASCII 字符组成的。例如，在发送数据 63 时，则需要发送‘6’和‘3’两个字符。由于数据中不包含‘：’、回车、换行三个字符，因此无需进行转义。

ASCII 模式数据帧格式如图 8-32 所示。

起始字符	Device Address	Function Code	Data	LRC check	结束字符
:	2 字符	2 字符	数个字符	2 字符	2 字符 <CR> <LF>

图 8-32　ASCII 模式数据帧格式

2. RTU 模式

RTU 模式采用的是二进制数据，即每个数据占 8 位，加上一个校验位，串口需要配置成 9b 模式。数据的取值范围为 0x00～0xFF，因此不能使用 ASCII 模式下的起始字符和结束字符。RTU 模式不同于 HDLC 和 PPP，采用了一种类似同步串行的方式，即：RTU 规定每次数据的传输结束，是以未再接到下一个字符间隔时间来判断。其规定为 3.5 字符的通信时间，例如：通信速率为 9600b/s、每个字符含 8b 再加上 1 个起始位及 1 个停止位后，一个字符为 10b。3.5 字符的通信时间为(3.5×10)/9600＝0.00365s，即在 3.65ms 内没有收到数据即认为是数据传输结束。

RTU 模式数据帧格式如图 8-33 所示。

开始间隔	Device Address	Function Code	Data	CRC check	结束间隔
T1-T2-T3-T4	8b	8b	Number of 8b	16b	T1-T2-T3-T4

图 8-33　RTU 模式数据帧格式

8.7　USART 函数库

8.7.1　寄存器定义

USART 寄存器结构 USART_TypeDef 的定义在 stm21L1xx.h 中：

```
typedef struct
{
    __IO uint16_t SR;                          //状态寄存器
    uint16_t RESERVED1;                        //保留
    __IO uint16_t DR;                          //数据寄存器
    uint16_t RESERVED2;                        //保留
    __IO uint16_t BRR;                         //波特率因子寄存器
    uint16_t RESERVED3;                        //保留
    __IO uint16_t CR1;                         //配置寄存器 1
    uint16_t RESERVED4;                        //保留
    __IO uint16_t CR2;                         //配置寄存器 2
    uint16_t RESERVED5;                        //保留
    __IO uint16_t CR3;                         //配置寄存器 3
    uint16_t RESERVED6;                        //保留
    __IO uint16_t GTPR;                        //保护时间和预分频寄存器
    uint16_t RESERVED7;                        //保留
} USART_TypeDef;
```

对于 STM32L152，通过如下定义可以确定 3 个 USART 的寄存器地址：

```
#define PERIPH_BASE ((uint32_t)0x40000000)
#define APB1PERIPH_BASE PERIPH_BASE
#define APB2PERIPH_BASE (PERIPH_BASE + 0x10000)
#define AHBPERIPH_BASE (PERIPH_BASE + 0x20000)
#define USART1_BASE (APB2PERIPH_BASE + 0x3800)
#define USART2_BASE (APB1PERIPH_BASE + 0x4400)
#define USART3_BASE (APB1PERIPH_BASE + 0x4800)
```

```
#define USART1 ((USART_TypeDef * ) USART1_BASE)
#define USART2 ((USART_TypeDef * ) USART2_BASE)
#define USART3 ((USART_TypeDef * ) USART3_BASE)
```

ST 提供了 NVIC 标准库函数，头文件位 stm32l1xx_uart.h，程序源代码位于 stm32l1xx_uart.c，USART_InitTypeDef 结构体用于串口的初始化配置，其定义如下：

```
typedef struct
{
    uint32_t USART_BaudRate;
    uint16_t USART_WordLength;
    uint16_t USART_StopBits;
    uint16_t USART_Parity;
    uint16_t USART_Mode;
    uint16_t USART_HardwareFlowControl;
} USART_InitTypeDef;
```

其中，USART_BaudRate 成员设置了 USART 传输的波特率，波特率的取值为 1200～4000000。

USART_WordLength 是一个帧中传输或者接收到的数据位数，其取值为：

- USART_WordLength_8b：8 位数据。
- USART_WordLength_9b：9 位数据。

USART_StopBits 定义了发送的停止位数目，其取值为：

- USART_StopBits_1：在帧结尾传输 1 个停止位。
- USART_StopBits_0.5：在帧结尾传输 0.5 个停止位。
- USART_StopBits_2：在帧结尾传输 2 个停止位。
- USART_StopBits_1.5：在帧结尾传输 1.5 个停止位。

USART_Parity 定义了奇偶模式，其取值为：

- USART_Parity_No：奇偶失能。
- USART_Parity_Even：偶模式。
- USART_Parity_Odd：奇模式。

USART_HardwareFlowControl 指定了硬件流控制模式使能还是失能，其取值为：

- USART_HardwareFlowControl_None：硬件流控制失能。
- USART_HardwareFlowControl_RTS：发送请求 RTS 使能。
- USART_HardwareFlowControl_CTS：清除发送 CTS 使能。
- USART_HardwareFlowControl_RTS_CTS：RTS 和 CTS 使能。

USART_Mode 指定了使能或者失能发送和接收模式，可同时使能发送或接收，其取值为：

- USART_Mode_Tx：发送使能。
- USART_Mode_Rx：接收使能。

USART 同步方式的时钟初始化配置结构体定义如下：

```
typedef struct
{
    uint16_t USART_Clock;
    uint16_t USART_CPOL;
    uint16_t USART_CPHA;
    uint16_t USART_LastBit;
} USART_ClockInitTypeDef;
```

USART_CLOCK 提示了 USART 时钟使能还是失能，其取值为：

- USART_Clock_Enable：时钟高电平活动。
- USART_Clock_Disable：时钟低电平活动。

USART_CPOL：USART_CPOL 指定了下 SLCK 引脚上时钟输出的极性，其取值为：

- USART_CPOL_High：时钟高电平。
- USART_CPOL_Low：时钟低电平。

USART_CPHA 指定了下 SLCK 引脚上时钟输出的相位，和 CPOL 位一起配合来产生不同的时钟/数据的采样关系，其取值为：

- USART_CPHA_1Edge：时钟第一个边沿进行数据捕获。
- USART_CPHA_2Edge：时钟第二个边沿进行数据捕获。

USART_LastBit 来控制是否在同步模式下，在 SCLK 引脚上输出最后发送的那个数据字（MSB)对应的时钟脉冲，其取值为：

- USART_LastBit_Disable：最后一位数据的时钟脉冲不从 SCLK 输出。
- USART_LastBit_Enable：最后一位数据的时钟脉冲从 SCLK 输出。

寄存器初始化结构体可以用来初始化同步串行模式或异步串行模式，每个成员的作用范围如表 8-5 所示。

表 8-5　同步和异步模式下的配置参数

成　　员	异 步 模 式	同 步 模 式
USART_BaudRate	X	X
USART_WordLength	X	X
USART_StopBits	X	X
USART_Parity	X	X
USART_HardwareFlowControl	X	X
USART_Mode	X	X
USART_Clock		X
USART_CPOL		X
USART_CPHA		X
USART_LastBit		X

8.7.2 USART 库函数

ST CMSIS 提供的 USART 主要 API 函数见表 8-6。

表 8-6 USART 主要函数

USART_DeInit	将外设 USARTx 寄存器重设为默认值
USART_Init	根据 USART_InitStruct 中指定的参数初始化外设 USARTx 寄存器
USART_StructInit	把 USART_InitStruct 中的每一个参数按默认值填入
USART_ClockInit	根据 USART_ClockInitStruct 进行时钟相关寄存器初始化
USART_ClockStructInit	把 USART_ClockInitStruct 中的每一个参数按默认值填入
USART_HalfDuplexCmd	使能或者失能 USART 半双工模式
USART_Cmd	使能或者失能 USART 外设
USART_DMACmd	使能或者失能指定 USART 的 DMA 请求
USART_OverSampling8Cmd	使能或失能 8 倍采样时钟
USART_OneBitMethodCmd	使能或失能过采样
USART_SendData	通过外设 USARTx 发送单个数据 USART
USART_ReceiveData	返回 USARTx 最近接收到的数据
USART_ITConfig	使能或者失能指定的 USART 中断
USART_GetFlagStatus	检查指定的 USART 标志位设置与否
USART_ClearFlag	清除 USARTx 的待处理标志位
USART_GetITStatus	检查指定的 USART 中断发生与否
USART_ClearITPendingBit	清除 USARTx 的中断待处理位
USART_SetAddress	设置 USART 节点的地址
USART_WakeUpConfig	选择 USART 的唤醒方式
USART_ReceiverWakeUpCmd	检查 USART 是否处于静默模式

1）USART_DeInit 函数

函数功能：将外设 USARTx 寄存器重设为复位值。

函数原型：void USART_DeInit(USART_TypeDef * USARTx)。

输入参数：USARTx：x 可以是 1,2,3,4,5,来选择 USART 外设。

示例：

```
USART_DeInit(USART1)/                              /将 USART1 复位
```

2）USART_Init 函数

函数功能：根据 USART_InitStruct 中指定的参数初始化外设 USARTx 寄存器。

函数原型：void USART_Init(USART_TypeDef * USARTx, USART_InitTypeDef * USART_InitStruct)。

输入参数 USARTx：x 可以是 1,2,3,4,5,来选择 USART 外设。

输入参数 USART_InitStruct：指向结构 USART_InitTypeDef 的指针,包含了外设 USART 的配置信息。

示例：

```
USART_InitTypeDef USART_InitStructure;
USART_InitStructure.USART_BaudRate = 9600;
USART_InitStructure.USART_WordLength = USART_WordLength_8b;
USART_InitStructure.USART_StopBits = USART_StopBits_1;
USART_InitStructure.USART_Parity = USART_Parity_Odd;
USART_InitStructure.USART_HardwareFlowControl =
                                           USART_HardwareFlowControl_RTS_CTS;
USART_InitStructure.USART_Mode = USART_Mode_Tx | USART_Mode_Rx;
USART_Init(USART1, &USART_InitStructure);
```

3）USART_StructInit 函数

函数功能：把 USART_InitStruct 中的每一个参数按默认值填入。

函数原型：void USART_StructInit(USART_InitTypeDef * USART_InitStruct)。

输入参数：USART_InitStruct：指向结构 USART_InitTypeDef 的指针,待初始化。

USART_InitStruct 默认值为：

```
USART_BaudRate:                  9600
USART_WordLength:                USART_WordLength_8b
USART_StopBits:                  USART_StopBits_1
USART_Parity:                    USART_Parity_No
USART_HardwareFlowControl:       USART_HardwareFlowControl_None
USART_Mode:                      USART_Mode_Rx | USART_Mode_Tx
```

示例：

```
USART_InitTypeDef USART_InitStructure;
USART_StructInit(&USART_InitStructure);
```

4）USART_ClockInit 函数

函数功能：根据 USART_ClockInitStruct 参数初始化 USARTx 的时钟配置。

函数原型：void USART_ClockInit(USART_TypeDef * USARTx, USART_ClockInitTypeDef * USART_ClockInitStruct)。

输入参数 USARTx,用于指定 USART,x 取值范围为 1,2,3。

输入参数 USART_ClockInitStruct,指向 USART_ClockInitTypeDef 的指针,包含了

USART 同步模式下的时钟配置信息。

示例：

```
USART_ClockInitTypeDef* USART_ClockInitStruct;
USART_ClockInitStructure.USART_Clock = USART_Clock_Disable;
USART_ClockInitStructure.USART_CPOL = USART_CPOL_High;
USART_ClockInitStructure.USART_CPHA = USART_CPHA_1Edge;
USART_ClockInitStructure.USART_LastBit = USART_LastBit_Enable;
USART_ClockInit(&USART_ClockInitStructure);
```

5）USART_ClockStructInit 函数

函数功能：把 USART_ClockInitStruct 中的每一个参数按默认值填入。

函数原型：void USART_ClockStructInit(USART_ClockInitTypeDef * SART_ClockInitStruct)。

输入参数 USART_ClockInitStruct：指向 USART_ClockInitTypeDef 的指针，待初始化。

USART_InitStruct 默认值为：

```
USART_Clock        USART_Clock_Disable
USART_CPOL         USART_CPOL_Low
USART_CPHA         USART_CPHA_1Edge
USART_LastBit      USART_LastBit_Disable
```

示例：

```
USART_ClockInitTypeDef* USART_ClockInitStruct;
USART_ClockStructInit(&USART_ClockInitStructure);
```

6）USART_OverSampling8Cmd 函数

函数功能：启用或禁用 USART 的 8 倍采样时钟模式。

函数原型：void USART_OverSampling8Cmd(USART_TypeDef * USARTx, FunctionalState NewState)。

输入参数 USARTx：用于指定待配置的串口，x 的取值范围为 1,2,3,4,5。

输入参数 NewState：用于使能或禁用 8 倍采样时钟，取值为 ENABLE 或 DISABLE。

该函数必须在 USART_Init 之前调用。示例如下：

```
USART_OverSampling8Cmd(USART1,ENABLE);
```

7）USART_OneBitMethodCmd 函数

函数功能：用于使能或禁用过采样功能。

函数原型：void USART_OneBitMethodCmd(USART_TypeDef * USARTx, FunctionalState NewState)。

输入参数 USARTx：用于指定待配置的串口，x 的取值范围为 1,2,3,4,5。

输入参数 NewState：用于使能或禁用过采样，取值为 ENABLE 或 DISABLE。

示例：

```
USART_OneBitMethodCmd(USART1,ENABLE);
```

8）USART_Cmd 函数

函数功能：使能或者失能 USART 外设。

函数原型：void USART_Cmd(USART_TypeDef * USARTx，FunctionalState NewState)。

输入参数 USARTx：x 可以是 1，2 或者 3，来选择 USART 外设。

输入参数 NewState：外设 USARTx 的新状态，参数取值为：ENABLE 或者 DISABLE。

示例：

```
USART_Cmd(USART1, ENABLE);
```

9）USART_DMACmd 函数

函数功能：使能或者失能指定 USART 的 DMA 请求。

函数原型：USART_DMACmd(USART_TypeDef * USARTx，uint16_t USART_DMAReq，FunctionalState NewState)。

输入参数 USARTx：用于指定待配置的串口，x 的取值范围为 1，2，3，4，5。

输入参数 USART_DMAreq：指定 DMA 请求，取值为：

- USART_DMAReq_Tx 发送 DMA 请求。
- USART_DMAReq_Rx 接收 DMA 请求。

输入参数 NewState：USARTx DMA 请求源的新状态，取值为 ENABLE 或者 DISABLE。

示例：

```
USART_DMACmd(USART2, USART_DMAReq_Rx | USART_DMAReq_Tx, ENABLE);
```

10）USART_SendData 函数

函数功能：发送一个字节的数据。

函数原型：void USART_SendData(USART_TypeDef * USARTx，uint16_t Data)。

输入参数 USARTx：用于指定待配置的串口，x 的取值范围为 1，2，3，4，5。

输入参数 Data：待发送的数据。

示例：

```
USART_SendData(USART3, 0x26);
```

11）USART_ReceiveData 函数

函数功能：返回 USARTx 最近接收到的数据。

函数原型：uint16_t USART_ReceiveData(USART_TypeDef * USARTx)。

输入参数 USARTx：用于指定待配置的串口，x 的取值范围为 1，2，3，4，5。

返回值：接收到的数据。

示例：

```
uint16_t RxData = USART_ReceiveData(USART2);
```

12）USART_ITConfig 函数

函数功能：使能或者失能指定的 USART 中断。

函数原型：void USART_ITConfig(USART_TypeDef * USARTx, uint16_t USART_IT, FunctionalState NewState)。

输入参数 USARTx：用于指定待配置的串口，x 的取值范围为 1,2,3,4,5。

输入参数 USART_IT：待使能或者失能的 USART 中断源，其取值为：

- USART_IT_PE　　奇偶错误中断
- USART_IT_TXE　　发送中断
- USART_IT_TC　　传输完成中断
- USART_IT_RXNE　　接收中断
- USART_IT_IDLE　　空闲总线中断
- USART_IT_LBD　　LIN 中断检测中断
- USART_IT_CTS　　CTS 中断
- USART_IT_ERR　　错误中断

输入参数 NewState：外设 USARTx 的新状态，参数取值为 ENABLE 或者 DISABLE。

示例：

```
USART_ITConfig(USART1, USART_IT_Transmit, ENABLE);
```

13）USART_GetFlagStatus 函数

函数功能：检查指定的 USART 标志位设置与否。

函数原型：FlagStatus USART_GetFlagStatus(USART_TypeDef * USARTx, uint16_t USART_FLAG)。

输入参数 USARTx：用于指定待配置的串口，x 的取值范围为 1,2,3,4,5。

输入参数 USART_FLAG：待检查的 USART 标志位，取值为：

- USART_FLAG_CTS　　CTS 标志位
- USART_FLAG_LBD　　LIN 中断检测标志位
- USART_FLAG_TXE　　发送数据寄存器空标志位
- USART_FLAG_TC　　发送完成标志位
- USART_FLAG_RXNE　　接收数据寄存器非空标志位
- USART_FLAG_IDLE　　空闲总线标志位
- USART_FLAG_ORE　　溢出错误标志位
- USART_FLAG_NE　　噪声错误标志位
- USART_FLAG_FE　　帧错误标志位
- USART_FLAG_PE　　奇偶错误标志位

返回值：待检查的标志位的状态，SET 或 RESET。

示例：

```
FlagStatus Status = USART_GetFlagStatus(USART1, USART_FLAG_TXE);
```

14）USART_ ClearFlag 函数

函数功能：清除 USARTx 的待处理标志位。

函数原型：void USART_ClearFlag(USART_TypeDef * USARTx，uint16_t USART_FLAG)。

输入参数 USARTx：用于指定待配置的串口，x 的取值范围为 1，2，3，4，5。

输入参数 USART_FLAG：待清除的 USART 标志位，取值和 USART_GetFlagStatus 的 USART_FLAG 相同。

示例：

```
USART_ClearFlag(USART1,USART_FLAG_OR);
```

15）USART_ GetITStatus 函数

函数功能：检查指定的 USART 中断发生与否。

函数原型：ITStatus USART_GetITStatus(USART_TypeDef * USARTx，uint16_t USART_IT)。

输入参数 USARTx：用于指定待配置的串口，x 的取值范围为 1，2，3，4，5。

输入参数 USART_IT：待检查的 USART 中断源，取值为：

- USART_IT_PE　　奇偶错误中断
- USART_IT_TXE　　发送中断
- USART_IT_TC　　发送完成中断
- USART_IT_RXNE　　接收中断
- USART_IT_IDLE　　空闲总线中断
- USART_IT_LBD　　LIN 中断探测中断
- USART_IT_CTS　　CTS 中断
- USART_IT_ORE　　溢出错误中断
- USART_IT_NE　　噪音错误中断
- USART_IT_FE　　帧错误中断

返回值：USART_IT 的新状态，SET 或 RESET。

示例：

```
ITStatus ErrorITStatus = USART_GetITStatus(USART1, USART_IT_ORE);
```

16）USART_ ClearITPendingBit 函数

函数功能：清除 USARTx 的中断待处理位。

函数原型：void USART_ClearITPendingBit(USART_TypeDef * USARTx，uint16_t

USART_IT)。

输入参数 USARTx：用于指定待配置的串口，x 的取值范围为 1,2,3,4,5。

输入参数 USART_IT：待检查的 USART 中断源，取值和 USART_ GetITStatus 函数 USART_IT 参数相同。

示例：

```
USART_ClearITPendingBit(USART1,USART_IT_OverrunError);
```

17）USART_SetAddress 函数

函数功能：设置 USART 设备的地址，用于多主机通信。

函数原型：void USART_SetAddress(USART_TypeDef * USARTx，uint8_t USART_Address)。

输入参数 USARTx：用于指定待配置的串口，x 的取值范围为 1,2,3,4,5。

输入参数 USART_Address：USART 设备的地址。

示例：

```
USART_SetAddress(USART2, 0x5);
```

18）USART_WakeUpConfig 函数

函数功能：选择 USART 的唤醒方式，用于多主机通信。

函数原型：void USART_WakeUpConfig(USART_TypeDef * USARTx，uint16_t USART_WakeUp)。

输入参数 USARTx：用于指定待配置的串口，x 的取值范围为 1,2,3,4,5。

输入参数 USART_WakeUp：USART 的唤醒方式，取值为：

- USART_WakeUp_IdleLine　　空闲总线唤醒
- USART_WakeUp_AddressMark　　地址标记唤醒

示例：

```
USART_WakeUpConfig(USART1, USART_WakeUpIdleLine);
```

19）USART_ReceiverWakeUpCmd 函数

函数功能：检查 USART 是否处于静默模式，用于多主机通信。

函数原型：void USART_ReceiverWakeUpCmd(USART_TypeDef * USARTx，FunctionalState Newstate)。

输入参数 USARTx：用于指定待配置的串口，x 的取值范围为 1,2,3,4,5。

输入参数 NewState：USART 静默模式的新状态，参数取值为 ENABLE 或者 DISABLE。

示例：

```
USART_ReceiverWakeUpCmd(USART3, DISABLE);
```

8.8　USART 案例

8.8.1　串口寄存器操作案例

1）串口初始化函数 USART_Init 的实现

```
void USART_Init(USART_TypeDef * USARTx, USART_InitTypeDef * USART_InitStruct)
{
  uint32_t tmpreg = 0x00, apbclock = 0x00;
  uint32_t integerdivider = 0x00;
  uint32_t fractionaldivider = 0x00;
  RCC_ClocksTypeDef RCC_ClocksStatus;
  tmpreg = USARTx- > CR2;
  tmpreg &= (uint32_t)~ ((uint32_t)USART_CR2_STOP); //清除 STOP 域
  //根据参数 USART_StopBits 设置 STOP 域 value
  tmpreg |= (uint32_t)USART_InitStruct- > USART_StopBits;
  //将配置参数写入控制寄存器 2
  USARTx- > CR2 = (uint16_t)tmpreg;
  tmpreg = USARTx- > CR1;
  //清除 M、PCE、PS、TE 和 RE 域
  tmpreg &= (uint32_t)~ ((uint32_t)CR1_CLEAR_MASK);
  //根据输入参数设置 M、PCE、PS、TE 和 RE 域
  tmpreg   |= (uint32_t)USART_InitStruct- > USART_WordLength
           | USART_InitStruct- > USART_Parity |USART_InitStruct- > USART_Mode;
  //将配置参数写入控制寄存器 1
  USARTx- > CR1 = (uint16_t)tmpreg;
  //配置控制寄存器 3
  tmpreg = USARTx- > CR3;
  //清除 CTSE 和 RTSE
  tmpreg &= (uint32_t)~ ((uint32_t)CR3_CLEAR_MASK);
  //根据输入参数 USART_HardwareFlowControl 配置 CTSE 和 RTSE 域
  tmpreg |= USART_InitStruct- > USART_HardwareFlowControl;
  USARTx- > CR3 = (uint16_t)tmpreg;
  //波特率寄存器配置,首先获取总线时钟
  RCC_GetClocksFreq(&RCC_ClocksStatus);
  //USART1 连接在 APB2,获取 APB2 总线时钟
  if (USARTx = = USART1)
      apbclock = RCC_ClocksStatus.PCLK2_Frequency;
  else //其余连接在 APB1,获取 APB1 总线时钟
```

```
        apbclock = RCC_ClocksStatus.PCLK1_Frequency;
    if ((USARTx->CR1 & USART_CR1_OVER8) != 0)
    //8倍波特率采样时钟配置下计算整数部分
     integerdivider= ((25* apbclock) / (2*  (USART_InitStruct->USART_BaudRate)));
    else //16倍波特率采样时钟配置下计算整数部分
    integerdivider= ((25* apbclock) / (4 *  (USART_InitStruct->USART_BaudRate)));
    tmpreg = (integerdivider / 100) << 4;
    //小数部分计算
    fractionaldivider = integerdivider - (100 *  (tmpreg >> 4));
    //8倍波特率采样时钟
    if ((USARTx->CR1 & USART_CR1_OVER8) != 0)
        tmpreg |= ((((fractionaldivider * 8) + 50) / 100)) & ((uint8_t)0x07);
    else //16倍波特率采样时钟
      tmpreg |= ((((fractionaldivider * 16) + 50) / 100)) & ((uint8_t)0x0F);
     //写入波特率寄存器
    USARTx->BRR = (uint16_t)tmpreg;
}
```

2）USART_SendData 函数的实现

```
void USART_SendData(USART_TypeDef*  USARTx, uint16_t Data)
{
  //数据最长为 9b,因此与 0x1FF进行位与操作
  USARTx->DR = (Data & (uint16_t)0x01FF);
}
```

8.8.2 串口配置基本流程

1）配置 I/O

```
//设置 Tx引脚为推拉输出模式
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9 | GPIO_Pin_2;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
GPIO_InitStructure.structure.GPIO_OType= GPIO_OType_PP;
GPIO_Init(GPIOA, &GPIO_InitStructure);
//设置 Rx引脚
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10 | GPIO_Pin_3;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
GPIO_InitStructure.GPIO_Pupd = GPIO_Pupd_Nopull;
GPIO_Init(GPIOA, &GPIO_InitStructure);
RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE);
```

2）配置 UART

```
void USART3_Configuration(void)
{
    USART_InitTypeDef USART_InitStructure;
    USART_InitStructure.USART_BaudRate = 115200;                    //设置波特率
    USART_InitStructure.USART_WordLength = USART_WordLength_8b;     //数据长度 8 位
    USART_InitStructure.USART_StopBits = USART_StopBits_1;          //一个停止位
    USART_InitStructure.USART_Parity = USART_Parity_No;             //无奇偶校验
    USART_InitStructure.USART_HardwareFlowControl
                    = USART_HardwareFlowControl_None;               //无非硬件流控制
    USART_InitStructure.USART_Mode= USART_Mode_Tx|USART_Mode_Rx;
    //允许接收和发送
    USART_Init(USART3,&USART_InitStructure);
    //配置接收中断
    USART_ITConfig(USART3, USART_IT_RXNE, ENABLE);
    //使能串口
    USART_Cmd(USART3, ENABLE);
}
```

3）配置 NVIC

```
void NVIC_Configuration(void)
//使能串口中断,同时要设置中断的优先级
NVIC_InitStructure.NVIC_IRQChannel= USART1_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority= 0;
NVIC_InitStructure.NVIC_IRQChannelSubPriority= 0;
NVIC_InitStructure.NVIC_IRQChannelCmd= ENABLE;
//使能串口中断
NVIC_Init(&NVIC_InitStructure)
```

4）中断函数 USART1_IRQHandler

```
Void USART3_IRQHandler(void)
{
    Unsigned char k= 0,bufl= 0;
    if(USART_GetITStatus(USART3,USART_IT_RXNE))
    {   //判断是否为接收数据中断
        bufl= USART_ReceiveData(USART3);
        USART_ClearITPendingBit(USART3,USART_FLAG_TC);
    }
}
```

8.8.3　PC 串口通信案例

1）查询方式发送数据

```
void USART_SendString(uint8_t * in)
```

```
{
    for(uint8_t i=0;in[i]!='\0';i++)
    {
        USART_Send(USART2,in[i]);
        while (USART_GetFlagStatus(USART2, USART_FLAG_TC) ==RESET);
    }
}
```

2）中断方式发送数据

```
uint8_t CmdBuffer[3]={'a','b','c'};
uint8_t TxIndex=0;
USART_ITConfig(USARTx, USART_IT_TXE, ENABLE);
while(1);
```

中断服务程序：

```
void USARTx_IRQHandler(void)
{
    if (USART_GetITStatus(USARTx, USART_IT_TXE) ==SET)
    {
        USART_SendData(USARTx, CmdBuffer[TxIndex++]);
        if (TxIndex ==0x03)
            USART_ITConfig(USARTx, USART_IT_TXE, DISABLE);
}
```

3）串口 printf 输出

```
#include <stdio.h>
int main(void)
{
  //配置串口
  USART_InitStructure.USART_BaudRate =115200;
  USART_InitStructure.USART_WordLength =USART_WordLength_8b;
  USART_InitStructure.USART_StopBits =USART_StopBits_1;
  USART_InitStructure.USART_Parity =USART_Parity_No;
  USART_InitStructure.USART_HardwareFlowControl=
                                                    USART_HardwareFlowControl_None;
  USART_InitStructure.USART_Mode =USART_Mode_Rx | USART_Mode_Tx;
  //TX RX GPIO 配置
  GPIO_InitTypeDef GPIO_InitStructure;
  RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOD, ENABLE);          //使能 UART 时钟
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART2, ENABLE);       //配置 USART_Tx 引脚
  GPIO_PinAFConfig(GPIOD, GPIO_Pin Source 5, GPIO_AF_USART2);  //配置 USART_Rx 引脚
  GPIO_PinAFConfig(GPIOD, GPIO_Pin Source 6, GPIO_AF_USART2);
```

```
  //配置 USART Tx I/O 复用推挽
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
  GPIO_Init(GPIOD, &GPIO_InitStructure);
  //配置 USART Rx I/O 复用推挽
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
  GPIO_Init(GPIOD, &GPIO_InitStructure);
  USART_Init(USART2, USART_InitStruct);                          //初始化串口
  USART_Cmd(USART2, ENABLE);                                     //使能串口
  //调用 printf 输出
  printf("\n\rUSART Printf Example\n\r");
  //判断 TC,等待 printf 传输完成
  while (USART_GetFlagStatus(USART2, USART_FLAG_TC) == RESET);
  while (1);
}
//重定向 printf 的输出
int fputc(int ch, FILE * f)
{
  USART_SendData(USART2, (uint8_t) ch);
  //等待数据寄存器为空,可继续写入下一个字符
  while (USART_GetFlagStatus(EVAL_COM1, USART_FLAG_TXE) == RESET);
  return ch;
}
```

8.8.4　状态机多字节数据帧发送和接收案例

数据通信规范采用 HDLC 规范进行数据发送,数据格式为:

7e	type	data	crc1	crc2	7e

发送数据定义:

```
typedef struct _TxMsg {
    uint8_t type;
    uint8_t data1;
    uint8_t data2;
} Tx_Msg ;
typedef Tx_Msg* Tx_MsgPtr;
```

接收数据定义：

```
typedef struct _MsgRcvEntry {
    uint8_t Length;
    Tx_Msg Msg;
    uint16_t CRC;
} Rx_Msg;
```

发送和接收数据缓冲区：

```
Tx_Msg gSendBuf= {65,1,2};
Rx_Msg gRcvBuf;
```

转义字符及数据中的 type 类型定义：

```
enum {
    HDLC_MTU         = (sizeof(Tx_Msg)),
    HDLC_FLAG_BYTE       = 0x7e,
    HDLC_CTLESC_BYTE     = 0x7d,
    PROTO_ACK            = 64,
    PROTO_PACKET_ACK     = 65,
    PROTO_PACKET_NOACK = 66,
    PROTO_UNKNOWN        = 255
};
```

发送状态机状态定义：

```
enum {
    TXSTATE_IDLE,
    TXSTATE_TYPE,
    TXSTATE_DATA,
    TXSTATE_ESC,
    TXSTATE_FCS1,
    TXSTATE_FCS2,
    TXSTATE_ENDFLAG,
    TXSTATE_FINISH,
    TXSTATE_ERROR
};
```

接收状态机状态定义：

```
enum {
    RXSTATE_NOSYNC,
    RXSTATE_TYPE,
    RXSTATE_DATA,
    RXSTATE_ESC
  };
```

状态机和变量初始化：

```
gRcvBuf.Length = 0;
uint8_t gTxState = TXSTATE_IDLE;
uint8_t gTxByteCnt = 0;
uint8_t gTxLength = 0;
uint16_t gTxRunningCRC = 0;
uint8_t* gpsend = (uint8_t*)(&gSendBuf);
uint8_t gRxState = RXSTATE_NOSYNC;
uint8_t gRxHeadIndex = 0;
uint8_t gRxTailIndex = 0;
uint8_t gRxByteCnt = 0;
uint16_t gRxRunningCRC = 0;
uint8_t* gpRxBuf = (uint8_t*)(& gRcvBuf.Tx_Msg);
```

主函数：

```
main(){
  //配置串口
  usart2config();
  //发送 0x7E,使能 TC 中断
  gTxLength = sizeof(gSendBuf);
  gTxState = TXSTATE_TYPE;
  USART_Send(USART2,HDLC_FLAG_BYTE);
  USART_IT_Config(USART2, USART_IT_TC,ENABLE);
  USART_IT_Config(USART2, USART_IT_RXNE,ENABLE);
  while(1);
}
```

中断处理函数：

```
void USARTx_IRQHandler(void){
  uint8_t nextByte= 0;
  if (USART_GetITStatus(USART2, USART_IT_TC) == SET){
    switch (gTxState) {
    case TXSTATE_TYPE:
      gTxState = TXSTATE_DATA;
      nextByte = * gpsend++;
      gTxRunningCRC = crcByte(gTxRunningCRC, nextByte);
      USART_Send(USART2, nextByte);
      gTxByteCnt++
      break;
    case TXSTATE_DATA:
      nextByte = * gpsend++;
      gTxRunningCRC = crcByte(gTxRunningCRC,nextByte);
```

```
        gTxByteCnt++;
        if (gTxByteCnt >= gTxLength)
          gTxState = TXSTATE_FCS1;
        TxArbitraryByte(nextByte);
        break;
      case TXSTATE_ESC:
        TxResult = USART_Send(USART2, (gTxEscByte ^ 0x20));
        gTxState = gPrevTxState;
        break;
      case TXSTATE_FCS1:
        nextByte = (uint8_t) (gTxRunningCRC & 0xff);                     // LSB
        gTxState = TXSTATE_FCS2;
        TxArbitraryByte(nextByte);
        break;
      case TXSTATE_FCS2:                                                  // MSB
        nextByte = (uint8_t) ((gTxRunningCRC >> 8) & 0xff);
        gTxState = TXSTATE_ENDFLAG;
        TxArbitraryByte(nextByte);
        break;
      case TXSTATE_ENDFLAG:
        gTxState = TXSTATE_FINISH;
        TxResult = USART_Send(USART2,HDLC_FLAG_BYTE);
        break;
      case TXSTATE_FINISH:
      case TXSTATE_ERROR:
      default:
        break;
      }
  }
  if (USART_GetITStatus(USART2, USART_IT_RXNE) == SET){
    uint8_t data =   USART_Receive(USART2);
    switch (gRxState) {
      case RXSTATE_NOSYNC:
      if ((data == HDLC_FLAG_BYTE) && (gRcvBuf.Length == 0)) {
            gRxByteCnt = gRxRunningCRC = 0;
            gRxState = RXSTATE_TPYE;
        }
        break;
      case RXSTATE_TPYE:
        * gpRxBuf++ = data;
        gRxRunningCRC = crcByte(gRxRunningCRC,data);
        gRxState = RXSTATE_DATA;
        gRxByteCnt++;
```

```
        break;
      case RXSTATE_DATA:
        if (gRxByteCnt > HDLC_MTU) {
            gRxByteCnt = gRxRunningCRC = 0;
            gRcvBuf.Length = 0;
            gRxState = RXSTATE_NOSYNC;
         }
         else if (data == HDLC_CTLESC_BYTE)
            gRxState = RXSTATE_ESC;
         else if (data == HDLC_FLAG_BYTE) {                       //收到结束字符
            if (gRxByteCnt >= 2) {
              uint16_t usRcvdCRC = (gpRxBuf[(gRxByteCnt-1)] & 0xff);
              usRcvdCRC = (usRcvdCRC << 8) | (gpRxBuf[(gRxByteCnt-2)] & 0xff);
                if (usRcvdCRC == gRxRunningCRC) {                 //校验
                gRcvBuf.Length = gRxByteCnt - 2;
                //PacketRcvd();判断接收数据进行下一步处理
                }
                 else
                    gRcvBuf.Length = 0;
             }
             else {                                               //不够数
                  gRcvBuf.Length = 0;
                  gRxState = RXSTATE_NOSYNC;
             }
             gRxByteCnt = gRxRunningCRC = 0;
         }
        else {
            * gpRxBuf++ = data;
            if (gRxByteCnt >= 2)
                gRxRunningCRC = crcByte(gRxRunningCRC,gpRxBuf[(gRxByteCnt-2)]);
            gRxByteCnt++;
      }
       break;
   }
   case RXSTATE_ESC:
      if (data == HDLC_FLAG_BYTE) {
        gRxByteCnt = gRxRunningCRC = 0;
        gMsgRcvTbl[gRxHeadIndex].Length = 0;
        gMsgRcvTbl[gRxHeadIndex].Token = 0;
        gRxState = RXSTATE_NOSYNC;
   }
   else {
      data = data ^ 0x20;
```

```
        gpRxBuf[gRxByteCnt] = data;
        if (gRxByteCnt >= 2) {
        gRxRunningCRC = crcByte(gRxRunningCRC,gpRxBuf[(gRxByteCnt-2)]);
      }
      gRxByteCnt++;
      gRxState = RXSTATE_INFO;
      }
      break;
    default:
        gRxState = RXSTATE_NOSYNC;
        break;
    }
      return SUCCESS;
}
//转义发送函数
uint8_t TxArbitraryByte(uint8_t Byte) {
      if ((Byte == HDLC_FLAG_BYTE) || (Byte == HDLC_CTLESC_BYTE)) {
        gPrevTxState = gTxState;
        gTxState = TXSTATE_ESC;
        gTxEscByte = Byte;
        Byte = HDLC_CTLESC_BYTE;
      }
        USART_Send(USART2,Byte);
      return 1;
    }
//crc函数
uint16_t crcByte(uint16_t crc,uint8_t data)
{
      //此处省略CRC计算函数
}
```

Chapter 9

第 9 章　IIC 总线

【导读】 IIC 总线常用于微控制器和数字外设芯片之间的连接，是一种典型的同步总线，本章首先介绍了 IIC 总线的时序，然后对 STM32L152 的 IIC 总线控制器的内部结构，寄存器以及不同模式的发送和接收配置流程进行了介绍，最后介绍了 CMSIS 提供的典型寄存器操作库函数。针对 IIC 总线时序传输，本章还对典型外设 Flash、温度传感器的操作时序进行了案例说明。

9.1　IIC 总线概述

IIC（Inter Integrated-Circuit，也记为 I^2C 或 I2C）总线是由 PHILIPS 公司 1982 年提出的一种用于连接微控制器和低速外设的短距离串行总线标准，通信速率从最初的 100kHz 到 2016 年 I2C 第六版高达 5MHz。I2C 是两线制总线，由一根时钟线 SCL 和一个数据线 SDA 组成。

I2C 是支持多主设备和多从设备的单工总线，总线硬件连接简单，将不同 I2C 设备的 SCL 和 SDA 直接相连即可，在多设备的使用中，I2C 总线采用地址寻址方法识别索要操作的设备，避免了片选寻址的弊端，从而使硬件系统扩展更为灵活。I2C 协议简单，适用于低成本芯片作为接口，典型的 I2C 接口外设有：串行存储器、低速 ADC 和 DAC、RTC 以及 LCD 控制等。在 I2C 的基础上，衍生出来的子标准有系统管理总线 SMBus、电源管理总线 PMBus、智能平台管理接口 IPMI、显示数据通道 DDC 和高级电信计算架构 ATCA 等。

I2C 是单工通信总线，由主设备和从设备构成，任何能够进行发送和接收的设备都可以成为主设备，主设备能够产生时钟，发起和结束一次数据传输。一般应用中，一个主设备控制多个从设备，I2C 支持多个主设备工作，在总线上可以有多个主设备发起通信，若产生冲突，通过总线仲裁进行解决，即在任何时间点只能有一个主设备处于通信状态。

I2C 总线的主要优势如下：

- 两线制总线，占用 I/O 数量少；
- 总线上的所有设备通过软件寻址，每个设备具有唯一的地址；
- 设备连接为主/从关系，主机可以是主发送器或主接收器；
- 通过冲突检测和仲裁机制支持多主机通信；
- 总线传输带有 ACK 和 NACK 应答，能保证数据传输的可靠性；

- 支持多种速率：标准模式(Standard Mode)100kb/s、快速模式(Fast Mode)400kb/s、增强快速模式(Fast Mode Plus)1Mb/s 和高速模式(High Speed Mode)3.4Mb/s；极速模式(Ultra-Fast Mode)，单向数据传输速率可达 5Mb/s。

I2C 只有数据和时钟两条线，在处理地址和应答时存在一定的开销，效率不如设备直接相连的 SPI 总线。I2C 的数据(SDA)和时钟(SCL)信号都是双向的，通过上拉电阻接到电源(如图 9-1 所示)。两根线都为高电平时，总线处于空闲状态(IDLE)。I2C 接口通过线与功能实现多设备的总线连接，总线的每个信号接口都包括一个开漏输出和输入缓冲器。开漏电路不能输出高电平，因此必须通过外接上拉电阻输出高电平。如果总线上有任何一个设备接口输出低电平，则整个总线的状态表现为低电平，体现出逻辑与的特点，即 I2C 的线与功能。线与功能的好处在于可以实现总线的仲裁控制。总线的控制权会交给最后一个输出低电平的设备，其他设备(输出为高)通过检测总线上的电平状态(表现为低)，对比与自己输出状态不一致，则自动退出对总线的控制请求。

开漏电路不适合长距离通信，信号线越长，信号的反射和振荡越强，从而影响总线的信号完整性，总线速度越快，对于线上干扰的要求越高，I2C 只适合电路板级的短距离通信。

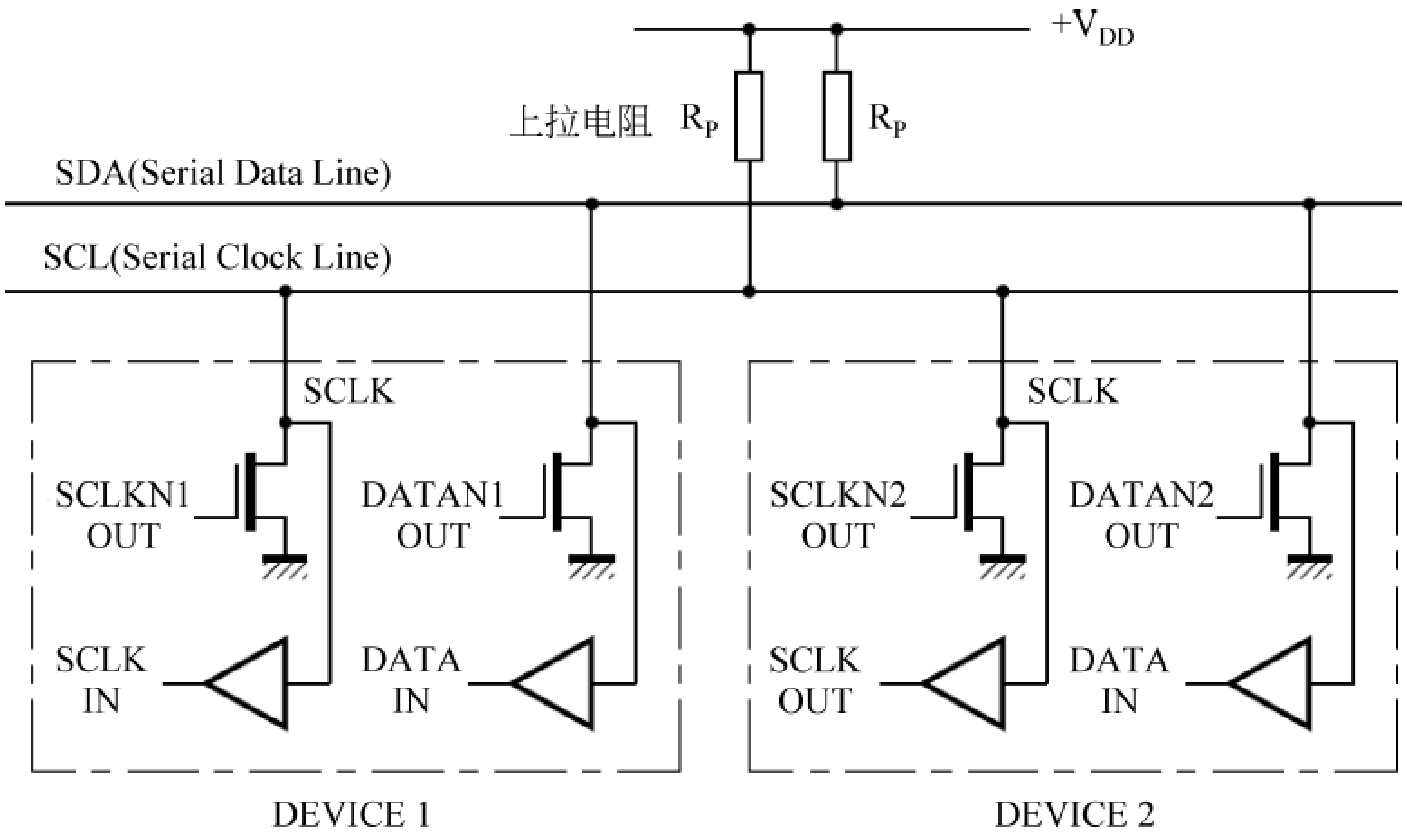


图 9-1　I2C 连接示意图

9.2　I2C 总线的基本操作

1. 数据有效性

I2C 协议一般采用 3V 或 5V 作为高电平 1，GND 作为低电平 0。每传输一比特数据 SDA，对应产生一个时钟脉冲 SCL。当 SCL 为高时，SDA 不允许变化；只有在 SCL 为低时，SDA 才可以变化，如图 9-2 所示。

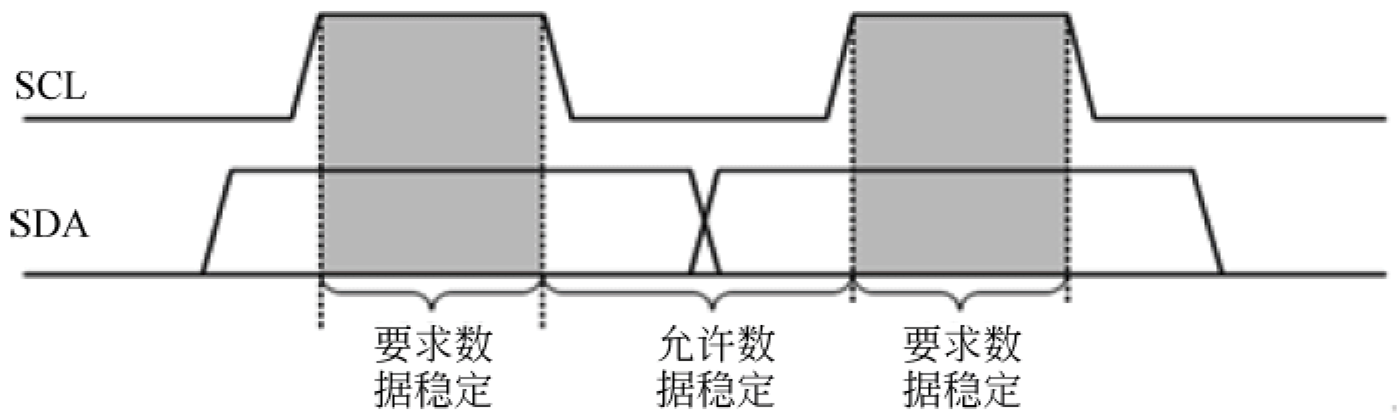


图 9-2　I2C 数据位的传输

2. 开始和结束条件

I2C 的两根线在空闲(IDLE)时均为高电平,SDA 和 SCL 的特定组合表示总线开始或结束一次数据传输,开始和结束的时序如图 9-3 所示。

(1) 开始条件 START(记为 S):SCL 为高时,SDA 由高变低。

(2) 结束条件 STOP(记为 P):SCL 为高时,SDA 由低变高。

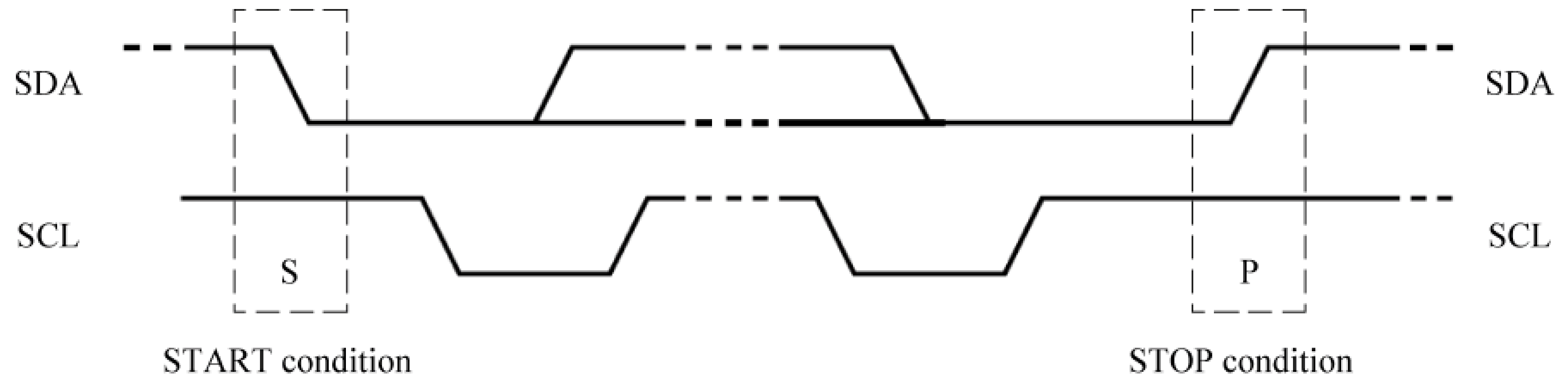


图 9-3　开始和结束条件

开始和结束条件总是由主机(Master)发起的。主机发出开始条件(START)后,总线处于忙的状态;主机发出结束条件(STOP)后,总线处于空闲状态(IDLE)。

在操作中,如果主机发出重复开始条件(Repeated START,记为 Sr)而非结束条件(STOP),则总线仍处于忙的状态。也就是说,重复开始条件(Sr)和开始条件(S)在功能上是相同的。判断开始或结束条件对于具有相应逻辑接口的设备相对简单,对于没有该接口的微控制器,需要在每个 SCL 周期内对 SDA 至少采样 2 次,才能正确检测到开始或结束条件。

3. 字节格式

SDA 上传输字节数据必须是 8 比特长度,每次传输不限定传输的字节数。每个字节(8 位)数据传送完毕后紧接着应答信号(第 9 位,Acknowledge Bit)。数据传输过程中,先发送高位(MSB),再发送低位(LSB),如图 9-4 所示。如果在数据传输过程中,从机如果没有准备好接收或发送下一个字节(比如内部中断需要处理等),它可以通过拉低 SCL 强制主机进入等待状态。直到从机释放 SCL,主机才开始下一个字节的发送或接收。

4. 应答

I2C 协议规定数据传输过程必须包含应答。接收器通过应答位(ACK bit)通知发送的字节已被成功接收,发送器可以进行下一个字节的传输。主机产生传输应答的第 9 个时钟。主机的发送器在应答时钟周期内释放对 SDA 的控制,这样从机接收器可以通过将 SDA 拉

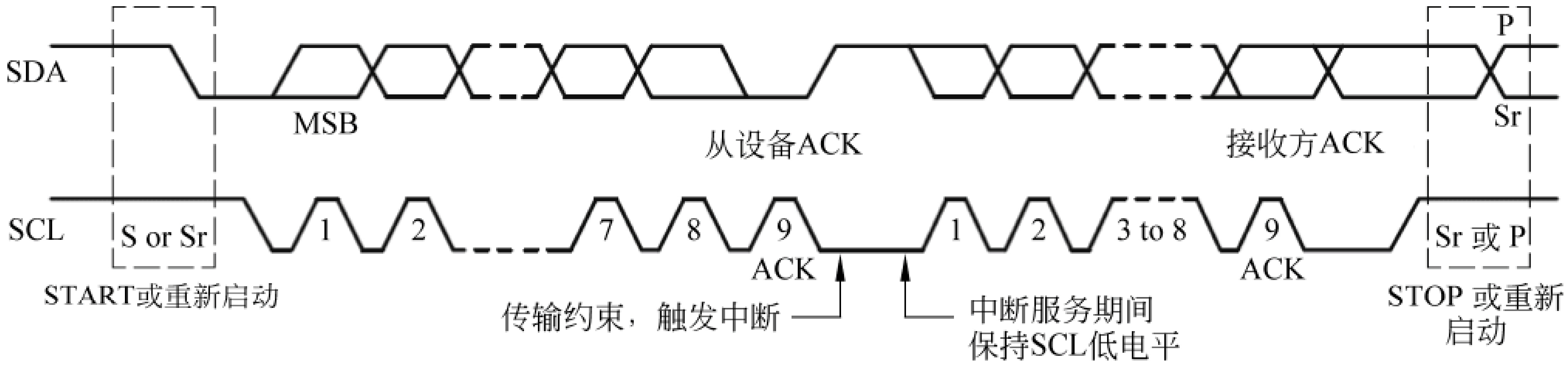


图 9-4　I2C 总线的数据传输

低通知发送器数据已被成功接收，如图 9-5 所示。

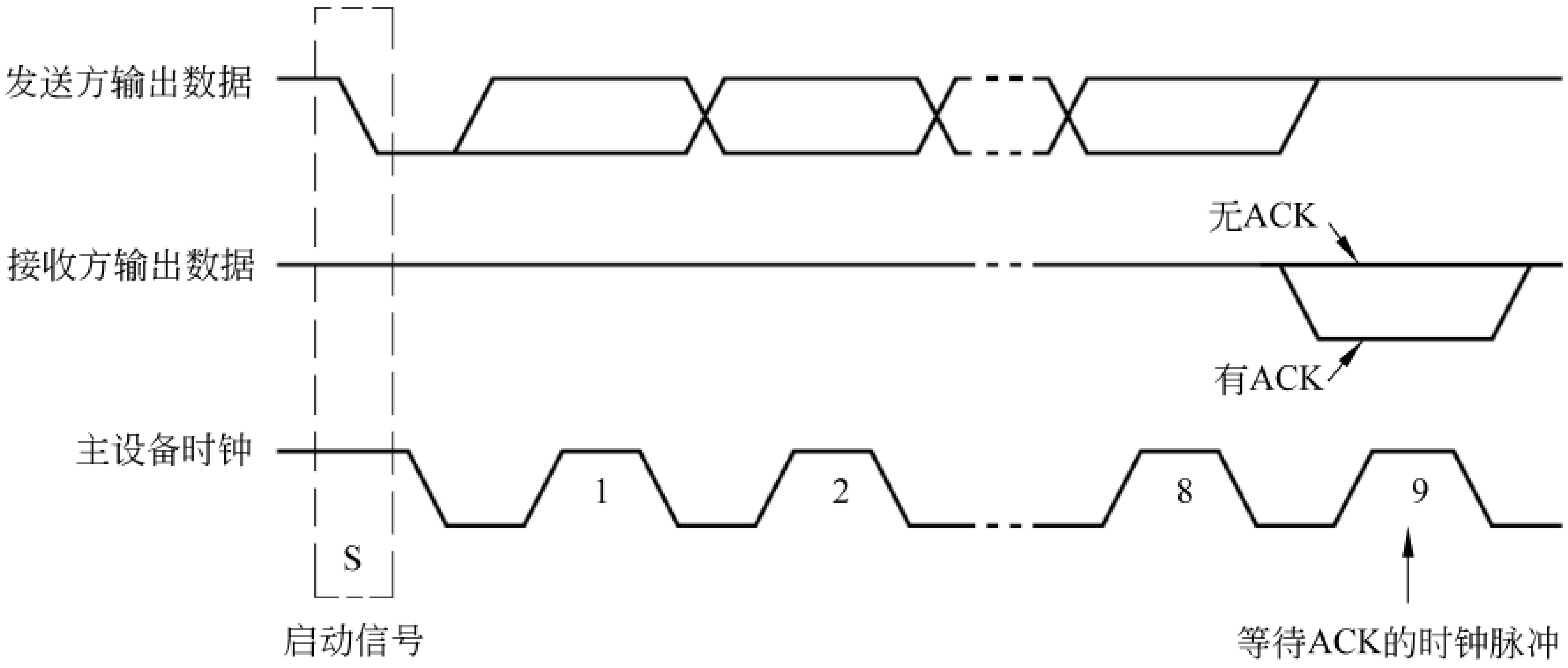


图 9-5　I2C 总线的数据应答

接收器发送 ACK 时，要保证 SCL 为高的同时，SDA 为低电平。如果在第 9 个时钟周期，SDA 为高，表明接收器无应答(NACK)，主机可以据此发出结束条件(STOP)命令结束此次传输，或发起重传请求(Repeated START)重新传输数据。以下情况可能导致无应答(NACK)：

- 总线上没有对应地址的接收器件。
- 接收器件没有准备好与主机的通信。
- 接收器件无法解析读取的数据。
- 接收器件无法收取更多的数据。

在连续传输中，主机发送器发送数据，从机接收数据并发送 ACK，主机的接收器在读取了从机发出的最后一个字节数据后，发出 NACK 通知从发送器释放数据线 SDA，主机随后发起结束(STOP)指令完成一次连续数据的传输。

图 9-6 表示主机作为发送器和接收器在写和读情况下的数据格式(ACK/NACK)。

I2C 的一大特点是可以在同一条总线上接多个主机。两个及以上的主机同时发起传输请求时，需要通过某种机制确定哪个主机获得总线的使用权；另外，每个主机都独立产生时钟，时钟速率可能千差万别，这也需要某种机制解决时钟速率不一致的问题。这种机制就是时钟同步(Clock Synchronization)和仲裁(Arbitration)。在单主机的 I2C 系统中，不需要时

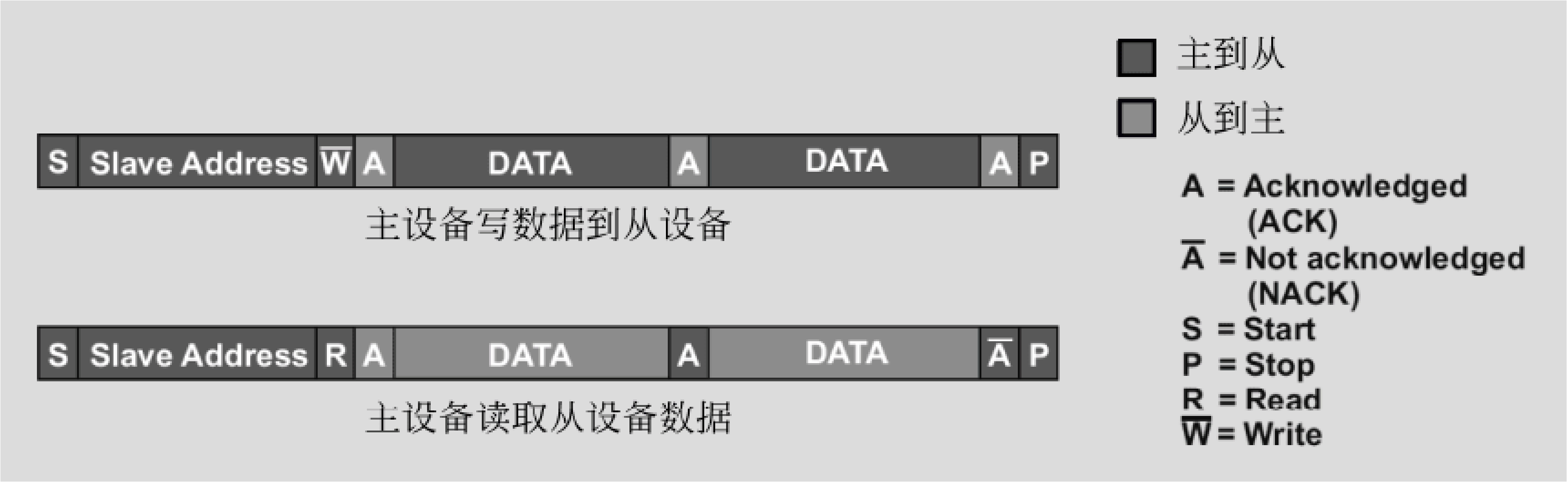


图 9-6　I2C 数据读写的格式

钟同步和仲裁。

5. 地址模式

I2C 的每一个从设备都具有一个设备地址，以便于在总线上连接多个从设备时对从设备进行寻址。I2C 规范有两种地址模式，7 位址模式和 10 位地址模式。

1）7 位地址系统

I2C 总线发送起始信号(START)后，发送的第一个字节由 7 位从设备地址和一位数据方向控制位 R/W 组成，如图 9-7 所示。

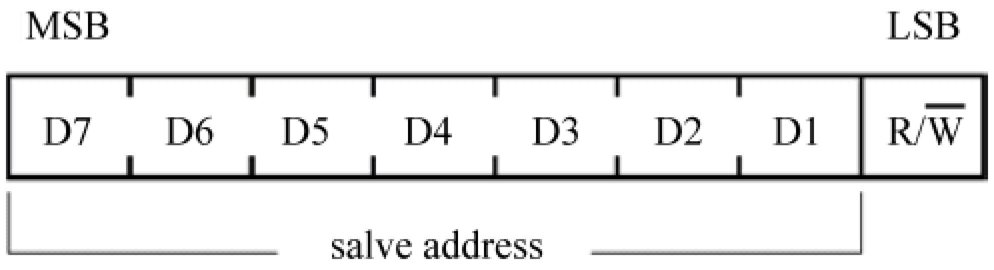


图 9-7　START 后的第一个字节格式

7 位的 I2C 设备地址由类型号和寻址码组成，其中 D7～D4 四位表示器件类型，器件类型是固定的，由 Philip 进行统一管理，D3～D1 是用户自定义的地址码，一般由电路连接不同的高低电平设置，由此可见，同一种类型的设备在 I2C 总线上最多只能连接 8 个。

D0 为数据方向控制位，D0＝1 表示主机读取从机设备，为 0 表示主机向从机发送数据。由读写位的特点可知对于 I2C 总线上的第一个字节，读操作都是奇数，写操作是偶数。

2）7 位地址的连续数据通信

主机发送开始条件后，可以连续进行多个字节的通信，直到主机发送结束条件(P)终止传输。主机也可以通过发起重复开始条件(Sr)进行一次新的传输，而不需要先产生结束条件(P)。

(1) 主机连续向从机发送数据，传输的方向不变。

如图 9-8 所示，主机发起开始条件，然后发送 7 位地址和一个低电平写控制位，立即改为从总线上读取状态，从机处于接收状态，读取总线地址，地址匹配后切换到发送模式向主机发送 ACK 确认，主机收到从机的应答(ACK)后，向从机发送第一个数据 DATA，并等待从机的 ACK 确认。主机发送 NACK，再发送结束条件(P)，结束本次传输。

(2) 主机连续读取从机数据，传输方向不变。

如图 9-9 所示，主机发起开始条件，然后发送 7 位地址和一个高电平读控制位，立即改

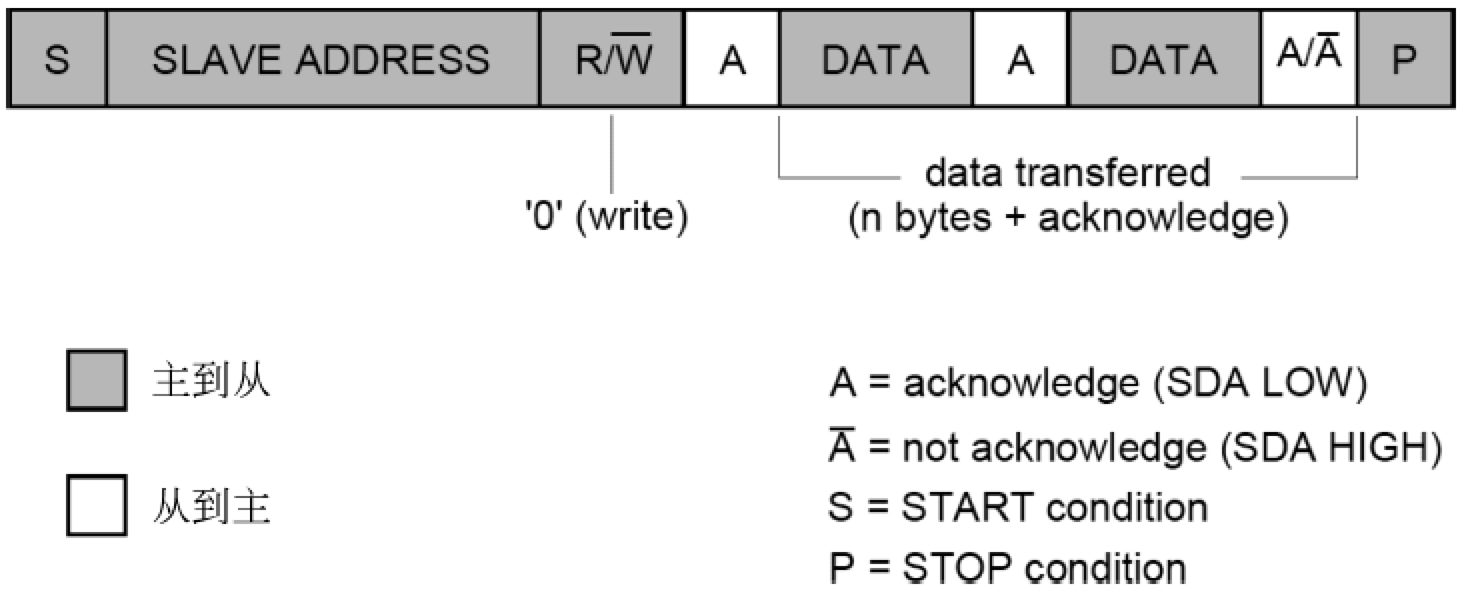


图 9-8　7 位地址主发从收模式

为从总线上读取状态，从机地址匹配后切换到发送模式向主机发送 ACK 确认，并开始发送第一个数据 DATA，主机收到从机的数据后，向从机发送 ACK 确认，并切换到接收模式接收从机的下一个数据。第一个 ACK 由从机发出，此后的 ACK 由主机发出。主机发送 NACK，再发送结束条件（P），结束本次传输。

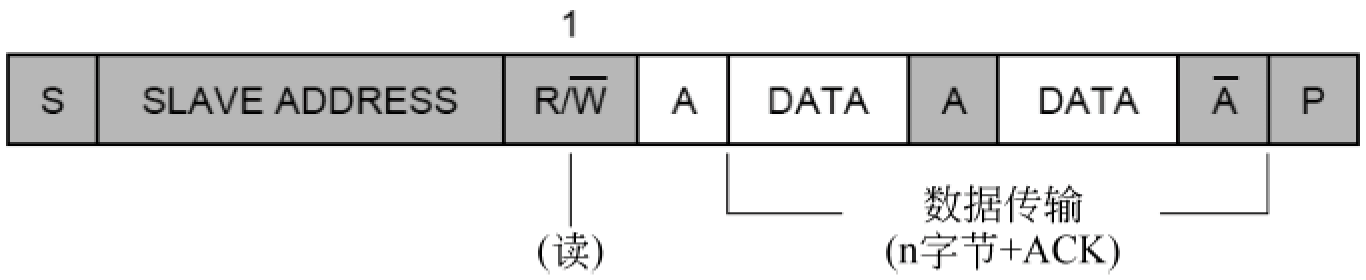


图 9-9　主机在第一个字节后立即读取从机内容

（3）读写混合模式，传输方向改变。

如图 9-10 所示，混合模式中，若要改变传输方向，则需要重新发送起始条件和从设备地址及读写控制位。

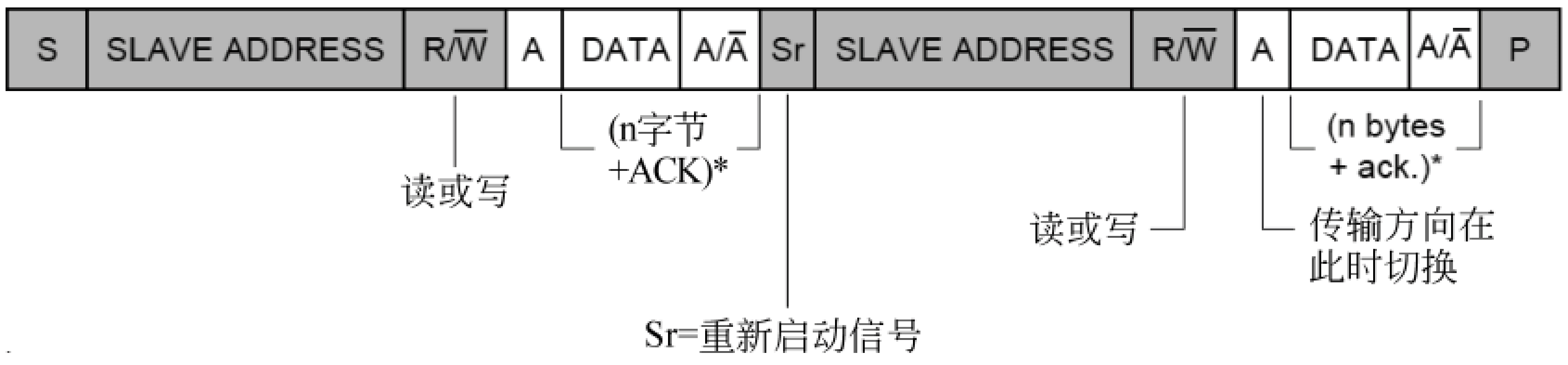


图 9-10　混合模式

3）10 位地址模式

采用 10 位地址系统扩充了 I2C 系统的地址范围。7 位和 10 位地址设备可以共存于同一个 I2C 总线系统，并且可工作在所有速度模式。目前使用 10 位地址系统的 I2C 设备不多。

10 位从机地址由两个字节 16 位组成，如图 9-11 所示，地址表示为 11110xx xxxxxxxx，发送地址时，先发送高 7 位（包含 10 位地址的最高两位）bits[15：9]，bit[8]用作读写方向控制位表明传输方向，此时总线上的所有从设备对比总线上第一个字节的前七位（1111 0XX）是否和自身地址一致，可能有一个以上设备会检测到地址匹配（因为只对比了 10 位地址的

最高 2 位)，它们都会产生响应 A1。收到 A1 后，主机发出 10 位地址的低 8 位(注意此次地址不包含读写方向控制位)bits[7：0]，所有上面响应的从机对比总线上第二个字节和它们各自地址的后八位(XXXX XXXX)是否一致。只有一个设备的地址匹配，并产生响应 A2。被寻址的从机一直受主机控制，直到 STOP 或 Sr 指向另外的地址。

图 9-11　主发送器寻址从接收器(10 位地址空间)

在混合传输下，主接收器改变读写方向时，无需每次都发送两次地址，如图 9-12 所示，在发送起始条件和 10 位地址之后，若需要改变读写方向，只需重新发送起始条件和第一个 7bit 地址及传输方向控制位即可，从机一直占用总线，直到接收到 STOP 或 Sr 指向另一个从机地址。

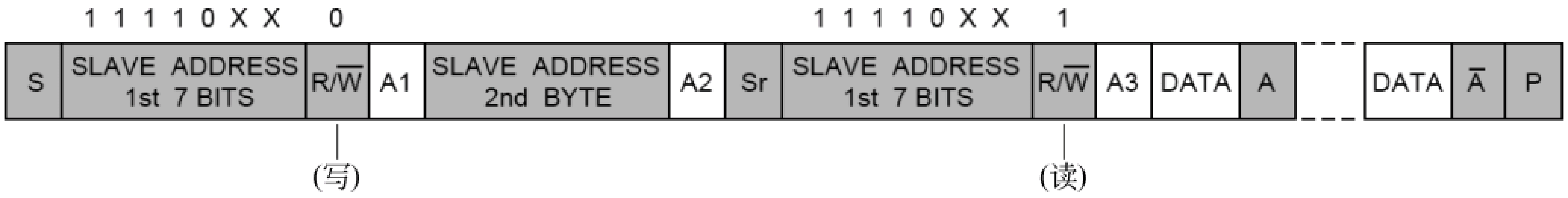


图 9-12　主接收器寻址从发送器(10 位地址空间)

从机地址中，0000000 0(R/W)表示通用广播地址，0000000 1(R/W)用于为不带 I2C 控制器的微处理器采用软件方式检测启动条件。

9.3　STM32L152 I2C 总线控制器

STM32L152 内部集成了两个独立的 I2C 总线控制器，I2C 总线控制器控制所有 I2C 总线特定的时序、协议、仲裁和定时，支持 CRC 码的生成和校验、系统管理总线 SMBus 和电源管理总线 PMBus 以及 DMA 传输，其主要特点为：

- 多主机功能：该模块既可做主设备也可做从设备。
- 可响应 2 个从地址的双地址能力。
- 产生和检测 7 位/10 位地址和广播呼叫。
- 支持标准速度(100kHz)和快速(高达 400kHz)两种模式。
- 支持多种状态标志和错误标志便于程序控制总线状态。
- 支持 2 个中断向量，用于地址/数据通信成功和错误处理。
- 可选的拉长时钟功能。
- 可配置的 PEC(信息包错误检测)的产生或校验。

• 兼容 SMBus 2.0。

I2C 控制器接收和发送数据，并将数据从串行转换成并行，或将并行转换成串行，接口通过数据引脚(SDA)和时钟引脚(SCL)连接，其内部结构如图 9-13 所示。

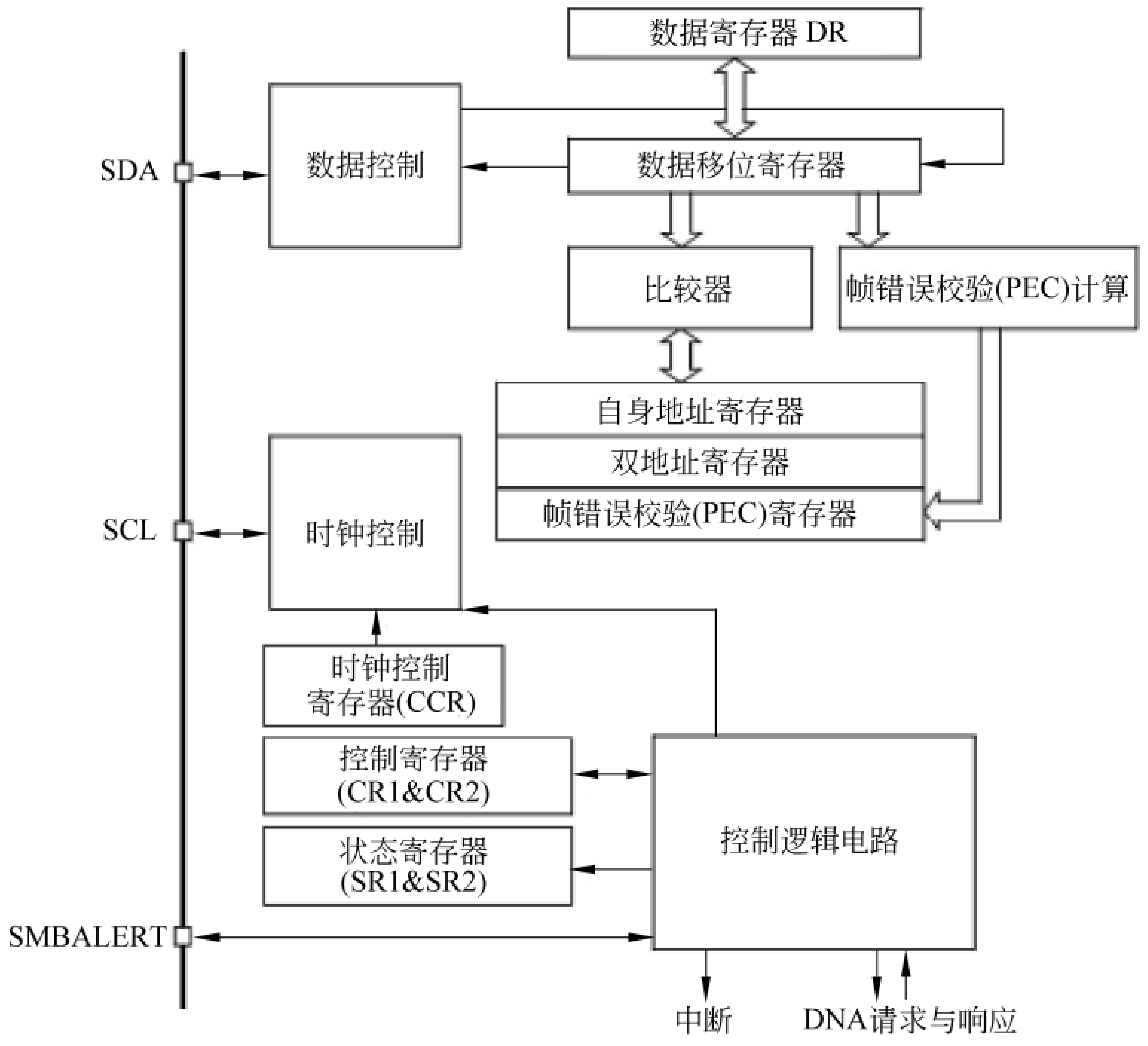


图 9-13　I2C 控制器内部结构

数据寄存器 DR 用于存储将要发送到 SDA 总线上的数据或者从 SDA 总线上读取的数据，数据通过一个移位寄存器进行发送或接收，发送数据时，数据寄存器的值被复制到移位寄存器，移位寄存器将 LSB 先送到总线上，直到 MSB 发送完成。接收时，当一个字节的所有位收到后，移位寄存器的数据被复制到数据寄存器 DR。

数据控制逻辑用于控制 SDA 的收发方向和 ACK 确认。比较器用于对从模式自身地址和主设备寻址发送的地址进行匹配，PEC 计算部分执行校验算法并和总线上的 PEC 数据进行比对。

控制逻辑电路通过时钟控制寄存器、控制寄存器对 I2C 的工作模式，时钟进行配置，维护状态寄存器，并向 CPU 发送中断和 DMA 请求。

I2C 控制器可以配置为下述 4 种模式中的一种运行：

• 从发送器模式。
• 从接收器模式。
• 主发送器模式。
• 主接收器模式。

MCU 启动后，I2C 默认地工作于从模式。I2C 控制器在接收到生成起始条件命令配置后自动地从从模式切换到主模式；当仲裁丢失或产生停止信号时，则从主模式切换到从模式。主模式时，I2C 接口启动数据传输并产生时钟信号。串行数据传输总是以起始条件开始并以停止条件结束。起始条件和停止条件都是在主模式下由软件控制产生。

从模式时，I2C 接口能识别它自己的地址（7 位或 10 位）和广播呼叫地址。软件能够控制开启或禁止广播呼叫地址的识别。

数据和地址按 8 位/字节进行传输，高位在前。跟在起始条件后的 1 或 2 个字节是地址（7 位模式为 1 个字节，10 位模式为 2 个字节）。地址只在主模式发送。在一个字节传输的 8 个时钟后的第 9 个时钟期间，接收器必须回送一个应答位（ACK）给发送器。软件可以开启或禁止应答（ACK），并可以设置 I2C 接口的地址（7 位、10 位地址或广播呼叫地址）。

9.4　I2C 寄存器描述

I2C 控制器的寄存器如表 9-1 所示。

表 9-1　I2C 控制器寄存器列表

寄存器名称	偏移量	功　　能	复　位　值
控制寄存器 1 I2C_CR1	0x00	起始信号、停止信号、使能等控制	0x0000 0000
控制寄存器 2 I2C_CR2	0x04	中断、DMA 和频率控制	0x0000 0000
自身地址寄存器 1 I2C_OAR1	0x08	从模式时的地址	0x0000 0000
自身地址寄存器 2 I2C_OAR2	0x0C	从模式时第二个地址	0x0000 0000
数据寄存器 I2C_DR	0x10	发送和接收数据	0x0000 0000
状态寄存器 1 I2C_SR1	0x14	接收、发送、地址匹配及错误状态	0x0000 0000
状态寄存器 1 I2C_SR2	0x18	主从、发送方向和忙状态指示	0x0000 0000
时钟控制寄存器 I2C_CCR	0x1C	SCLK 参数配置	0x0000 0000
Trise 寄存器 I2C_TRISE	0x20	电平边沿变化速度	0x0000 0000

1. 控制寄存器 1（I2C_CR1）

控制寄存器的有效域定义如图 9-14 所示。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SWRST	保留	ALERT	PEC	POS	ACK	STOP	START	NO STRETCH	ENGC	ENPEC	ENARP	SMB TYPE	保留	SMBUS	PE
rw	res	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	res	rw	rw

图 9-14　控制寄存器 CR1

SWRST（Software Reset）：软件复位，该位被置 1 时，I2C 处于复位状态。

ALERT：SMBus 提醒，用于 SMBus 总线。

PEC(Packet Error Checking)：数据包出错检测。

POS(Position)：应答 ACK 的发送时机和 PEC 位置指示(用于数据接收)。

ACK：应答使能，0 表示无应答返回，1 表示在接收到一个字节后返回一个应答。

STOP：停止条件产生，该位置 1 表示产生一个停止条件，主模式下，在当前字节传输或在当前起始条件发出后产生停止条件，从模式下，在当前字节传输或释放 SCL 和 SDA 线后产生停止条件。该位可由软件设置，当检测到停止条件或超时错误时，硬件将自动清除该位。

START：起始条件产生，该位置 1 表示产生一个起始条件，主模式下设置该位为 1 将重复产生起始条件，从模式下，当总线空闲时产生起始条件。该位可由软件设置，当起始条件发出后该位由硬件自动清除。

NOSTRETCH：禁止时钟延长(从模式)，该位用于当 ADDR 或 BTF 标志被置位，在从模式下禁止时钟延长，直到它被软件复位。

ENGC(General Call Enable)：广播呼叫使能，置 0 表示禁止广播呼叫。以非应答响应地址 00h，置 1 表示允许广播呼叫. 以应答响应地址 00h。

ENPEC(PEC Enable)：PEC 使能，置 0 禁止 PEC 计算，置 1 开启 PEC 计算。

ENARP(ARP Enable)：ARP 使能 置 0 禁止 ARP，置 1 使能 ARP。ARP 是 SMBus 功能。

SMBTYPE(SMBus Type)：SMBus 类型 0 表示 SMBus 设备，1 表示 SMBus 主机。

SMBUS(SMBus Mode)：SMBus 模式，置 1 表示启用 SMBus 模式，置 0 表示 I2C 模式。

PE(Peripheral enable)：I2C 模块使能，置 1 表示启用 I2C 模块。

2. 控制寄存器 2(I2C_CR2)

控制寄存器 2 的有效域定义如图 9-15 所示。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留			LAST	DMAEN	ITBUF EN	ITEVT EN	ITERR EN	保留		FREQ[5:0]					
			rw	rw	rw	rw	rw			rw	rw	rw	rw	rw	rw

图 9-15　控制寄存器 CR2

LAST：DMA 最后一次传输，置 1 表示下一次 DMA 的 EOT 是最后的传输。

DMAEN(DMA Request Enable)：DMA 请求使能，0 表示禁止 DMA 请求，1 表示当 TxE=1 或 RxNE =1 时，允许 DMA 请求。

ITBUFEN(Buffer Interrupt Enable)：缓冲器中断使能，0 表示当 TxE=1 或 RxNE=1 时，不产生任何中断，1 表示当 TxE=1 或 RxNE=1 时，产生事件中断。

ITEVTEN(Event Interrupt Enable)：事件中断使能，置 0 禁止事件中断，置 1 允许事件中断。在下列条件下，将产生该中断：

- SB=1(主模式)；
- ADD10=1(主模式)；
- 如果 ITBUFEN=1，TxE 事件为 1；
- ADDR=1(主/从模式)
- STOPF=1(从模式)
- 如果 ITBUFEN=1，RxNE 事件为 1

-BTF=1,但是没有 TxE 或 RxNE 事件;

ITERREN(Error Interrupt Enable):出错中断使能 0 表示禁止出错中断,1 表示允许出错中断。如果 BERR=1、ARLO=1、AF=1、OVR=1、PECERR=1、TIMEOUT、SMBAlert=1,则产生中断。

FREQ[5:0]:I2C 模块时钟频率,时钟频率允许的范围为 2M~50MHz,但不能超过总线时钟。FREQ 取 000000 和大于 110010 的值表示禁用,000010~110010 分别表示 2MHz,3MHz,…,50MHz。

3. 自身地址寄存器 1(I2C_OAR1)

自身地址寄存器 1 的有效域定义如图 9-16 所示。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD MODE	保留	保留				ADD[9:8]		ADD[7:1]							ADD0
rw	res	res				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 9-16　自身地址寄存器 OAR1

ADDMODE(Address Mode):从模式下寻址模式,0 表示 7 位地址,1 表示 10 位地址。

ADD[9:8]:从设备地址 9~8 位:7 位地址模式时无效,10 位地址时为地址的 9~8 位。

ADD[7:1]:从设备地址 7~1 位。

ADD0:从设备地址 0 位,7 位地址模式时用于表示读写方向位,10 位地址模式时为地址第 0 位。

4. 自身地址寄存器 2(I2C_OAR2)

自身地址寄存器 2 的有效域定义如图 9-17 所示。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留								ADD2[7:1]							ENDUAL
res								rw	rw	rw	rw	rw	rw	rw	rw

图 9-17　自身地址寄存器 OAR2

ADD2[7:1]:从设备地址 7~1 位。

ENDUAL(Dual Address Mode Enable):双地址模式使能位,置 0 表示在 7 位地址模式下,只有 OAR1 被识别,置 1 表示在 7 位地址模式下,OAR1 和 OAR2 都被识别。

5. 数据寄存器(I2C_DR)

数据寄存器有效域定义如图 9-18 所示。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留								DR[7:0]							
res								rw	rw	rw	rw	rw	rw	rw	rw

图 9-18　数据寄存器 DR

DR[7:0]:8 位数据寄存器,用于存放接收到的数据或放置用于发送到总线的数据,发送器模式:当写一个字节至 DR 寄存器时,自动启动数据传输。一旦传输开始(TxE=1),

如果能及时把下一个需传输的数据写入 DR 寄存器，I2C 模块将保持连续的数据流。接收器模式：接收到的字节被复制到 DR 寄存器(RxNE=1)。在接收到下一个字节(RxNE=1)之前读出数据寄存器，即可实现连续的数据传送。在从模式下，地址不会被复制进数据寄存器 DR，且硬件不管理写冲突，如果 TxE=0，仍能写入数据寄存器，此时数据会发生错误。

6. 状态寄存器 1(I2C_SR1)

状态寄存器 1 的有效域定义如图 9-19 所示。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMB ALERT	TIME OUT	保留	PEC ERR	OVR	AF	ARLO	BERR	TxE	RxNE	保留	STOPF	ADD10	BTF	ADDR	SB
rc w0	rc w0	res	rc w0	rc w0	rc w0	rc w0	rc w0	r	r	res	r	r	r	r	r

图 9-19 状态寄存器 SR1

SMBALERT (SMBusAlert)：SMBus 告警提醒。

TIMEOUT：超时或 Tlow 错误，该位为 1 表明 SCL 处于低已达到 25ms(超时)，或者主机低电平累积时钟扩展时间超 10ms(Tlow：mext)，或从设备低电平累积时钟扩展时间超过 25ms。从模式下该位为 1 时从设备复位，硬件释放总线，主模式下设置该位，硬件发出停止条件。该位可由软件写 0 清除，或在 PE=0 时由硬件自动清除。

PECERR(PEC ERROR)：在接收时发生 PEC 错误。0 表示无 PEC 错误，接收到 PEC 后接收器返回 ACK(如果 ACK=1)；1 表示有 PEC 错误，接收到 PEC 后接收器返回 NACK(不管 ACK 是什么值)。该位可由软件写 0 清除，或在 PE=0 时由硬件自动清除。

OVR：过载/欠载标志，1 表示出现出现过载/欠载。当 NOSTRETCH=1，在接收模式中当收到一个新的字节时(包括 ACK 应答脉冲)，数据寄存器里的内容还未被读出，则新接收的字节将丢失；在发送模式中当要发送一个新的字节时，却没有新的数据写入数据寄存器，同样的字节将被发送两次，在这两种情况下，该位被硬件置自动置 1。该位可由软件写 0 清除，或在 PE=0 时由硬件自动清除。

AF(Acknowledge Failure)：应答失败，该位为 1 表示应答失败。当没有返回应答 ACK 时，硬件将该位自动置为 1。该位可由软件写 0 清除，或在 PE=0 时由硬件自动清除。

ARLO(Arbitration Lost)：仲裁丢失(主模式)，该位为 1 表明检测到仲裁丢失。当接口失去对总线的控制给另一个主机时，硬件将置该位为 1。该位可由软件写 0 清除，或在 PE=0 时由硬件自动清除。在 ARLO 事件之后，I2C 接口自动切换回从模式(M/SL=0)。

BERR(Bus Error)：总线出错，该位为 1 表示起始条件或停止条件出错，当 IIC 总线检测到错误的起始或停止条件，硬件将该位置 1。该位可由软件写 0 清除，或在 PE=0 时由硬件自动清除。

TxE(Transmit Data Register Empty)：数据发送寄存器为空标志，0 表示非空，1 表示空。在发送数据时，数据寄存器为空时该位被置 1，在发送地址阶段不设置该位。软件写数据到 DR 寄存器可清除该位；或者在发生一个起始或停止条件后、PE=0 时由硬件自动清除。如果收到一个 NACK，或下一个要发送的字节是 PEC(PEC=1)，该位不被置位。

RxNE(Receive Data Register Empty)：接收数据寄存器非空标志，1 表示非空，0 表示

空。在接收时，当数据寄存器不为空时该位被置1。在接收地址阶段，该位不被置位。软件对数据寄存器的读写操作清除该位，或当 PE=0 时由硬件清除。

STOPF(Stop Detection Flag)：从模式停止条件检测位，该位置1表示检测到停止条件。在一个应答之后(如果 ACK=1)，当从设备在总线上检测到停止条件时，硬件将该位置1。软件读取 SR1 寄存器后，对 CR1 寄存器的写操作将清除该位，或当 PE=0 时，硬件清除该位。在收到 NACK 后，STOPF 位不被置位。

ADD10：主模式10位地址序列发送标志，该位置1表示主设备已经将从设备地址的第一个字节发送出去。在10位地址模式下，当主设备已经将地址的第一个字节发送出去时，硬件将该位置1。软件读取 SR1 寄存器后，对 CR1 寄存器的写操作将清除该位，或当 PE=0 时，硬件清除该位。收到一个 NACK 后，ADD10 位不被置位。

BTF(Byte Transfer Finished)：字节发送结束。0 表示字节发送未完成，1 表示字节发送结束。当 NOSTRETCH=0 时，在下列情况下硬件将该位置1：

- 当收到一个新字节(包括 ACK 脉冲)且数据寄存器还未被读取(RxNE=1)。
- 当一个新数据将被发送且数据寄存器还未被写入新的数据(TxE=1)。

软件读取 SR1 寄存器后，对数据寄存器的读或写操作将清除该位；或在传输中发送一个起始或停止条件后，或当 PE=0 时，由硬件清除该位。在收到一个 NACK 后，BTF 位不会被置位。

ADDR：主模式地址发送标记/从模式地址匹配标记。从模式中，该位置0表示地址不匹配或没有收到地址，当收到的地址与 OAR 寄存器中的地址匹配时硬件自动将该位置1；主模式中，该位为0表示地址发送没有结束，1表示地址发送结束。10位地址模式时，当收到地址的第二个字节的 ACK 后该位被置1；7位地址模式时，当收到地址的 ACK 后该位被置1。在软件读取 SR1 寄存器后，对 SR2 寄存器的读操作将清除该位，或当 PE=0 时，由硬件清除该位。

SB(Start Bit)：主模式起始位标志，0 表示未发送起始条件，1 表示起始条件已发送。当发送出起始条件时该位被置1。软件读取 SR1 寄存器后，写数据寄存器的操作将清除该位，或当 PE=0 时，硬件清除该位。

7. 状态寄存器 2 (I2C_SR2)

状态寄存器2的有效域定义如图 9-20 所示。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PEC[7:0]								DUALF	SMB HOST	SMB DEFAULT	GEN CALL	保留	TRA	BUSY	MSL
r	r	r	r	r	r	r	r	r	r	r	r	res	r	r	r

图 9-20 状态寄存器 SR2

PEC[7:0](Packet Error Checking)：数据包出错检测，当 ENPEC=1 时，PEC[7:0]存放内部的 PEC 的值。

DUALF(Dual Flag)：从模式双地址标志，该位为0表示接收到的地址与 OAR1 内的

内容相匹配，1 表示接收到的地址与 OAR2 内的内容相匹配。在产生一个停止条件或一个重复的起始条件时，或 PE=0 时，硬件将该位清除。

SMBHOST：SMBus 主机地址标志。

SMB DEFAULT：SMBus 默认地址标志。

GENCALL：广播呼叫地址标志。

TRA(Transmit/Receive)：发送/接收标志，0 表示接收到数据，1 表示数据已发送。该位根据地址字节的 R/W 位来设定。在检测到停止条件(STOPF=1)、重复的起始条件或总线仲裁丢失(ARLO=1)后，或当 PE=0 时，硬件将其清除。

BUSY：总线忙标志，0 表示在总线上无数据通信，1 表示总线上正在进行数据通信。在检测到 SDA 或 SCl 为低电平时，硬件将该位置 1，当检测到停止条件时，硬件将该位清除。

MSL(Master Slave)：主从模式，0 表示从模式，1 表示主模式，当总线配置为主模式(SB=1)时，硬件将该位置位；当总线上检测到一个停止条件、仲裁丢失(ARLO=1 时)或当 PE=0 时，硬件清除该位。

8. 时钟控制寄存器(I2C_CCR)

时钟控制寄存器的有效域定义如图 9-21 所示。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
F/S	DUTY	保留		CCR[11:0]											
rw	rw			rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 9-21 时钟控制寄存器 CCR

F/S(Fast/Standard Mode Selection)：I2C 主模式选项，0 表示标准模式，1 表示快速模式。

DUTY：快速模式时的占空比，0 表示快速模式下 Tlow/Thigh = 2，1 表示快速模式下 Tlow/Thigh = 16/9。

CCR[11：0]：时钟系数，用于设置主模式下的 SCL 时钟。在 I2C 标准模式或 SMBus 模式下：$T_{high}=CCR\times T_{PCLK1}$，$T_{low}=CCR\times T_{PCLK1}$；在 I2C 快速模式下，如果 DUTY = 0，$T_{high}=CCR\times T_{PCLK1}$，$T_{low}=2\times CCR\times T_{PCLK1}$；如果 DUTY = 1，$T_{high}=9\times CCR\times T_{PCLK1}$，$T_{low}=16\times CCR\times T_{PCLK1}$。CCR 允许设定的最小值为 0x04，在快速 DUTY 模式下允许的最小值为 0x01；只有在关闭 I2C 时(PE = 0)才能设置 CCR 寄存器，且 F_{pclk} 应当是 10MHz 的整数倍，这样可以正确产生 400kHz 的快速时钟。

9. TRISE 寄存器(I2C_TRISE)

上升时间寄存器的有效域定义如图 9-22 所示。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留										TRISE[5:0]					
res										rw	rw	rw	rw	rw	rw

图 9-22 Trise 寄存器

TRISE[5：0](Time of Rise)：在快速/标准模式下的最大上升时间，这些位必须设置为 I2C 总线规范里给出的最大的 SCL 上升时间，增长步幅为 1。例如：标准模式中最大允许 SCL 上升时间为 1000ns。如果在 I2C_CR2 寄存器中 FREQ[5：0]中的值等于 0x08，则 TPCLK1 =125ns，故 TRISE[5：0]中必须写入 09h(1000ns/125 ns = 8+1)。只有当 I2C 被禁用(PE=0)时，才能设置 TRISE[5：0]。

9.5　I2C 数据通信流程

9.5.1　I2C 从模式通信

默认情况下，I2C 接口总是工作在从模式。由从模式切换到主模式，需要产生一个起始条件。为了产生正确的时序，必须在 I2C_CR2 寄存器中设定该模块的输入时钟。输入时钟的频率在标准模式下至少是 2MHz，快速模式下至少 4MHz。

一旦检测到起始条件，I2C SDA 线上接收到的地址被送到移位寄存器。然后与芯片自己的地址 OAR1 和 OAR2(当 ENDUAL=1 时)或者广播呼叫地址(当 ENGC=1 时)相比较。

如果头段或地址不匹配，I2C 将其忽略并等待另一个起始条件。如果地址匹配，I2C 接口产生以下时序：

- 若启用 ACK(寄存器 I2C_CR1 的 ACK=1)，则产生一个应答脉冲。
- 硬件自动设置 I2C_SR1 寄存器的 ADDR 位为 1，如果 I2C_CR2 中配置了 ITEVFEN=1，则产生一个中断。
- 如果 I2C_OAR2 寄存器的 ENDUAL 位为 1，软件必须读 I2C_SR2 寄存器的 DUALF 位，以确认响应了哪个从地址。

在从模式下 I2C_SR2 的 TRA 位指示当前是处于接收器模式还是发送器模式。

1. 从模式下的发送

从模式下的发送指的是接收主设备的读请求，从设备向主设备主动发送数据。从模式下的发送流程如图 9-23 所示，图中 EVx 表示 I2C 通信过程中产生的事件，若设置了 ITEVFEN=1，则会产生一个中断。

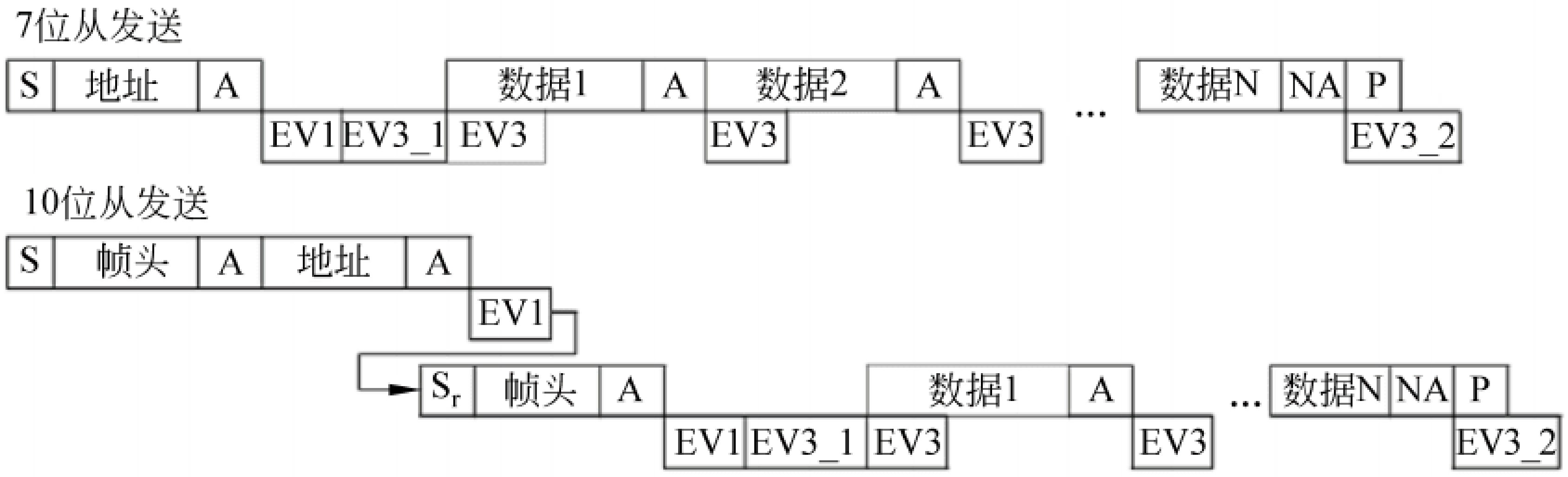


图 9-23　从模式发送时序

从发送器在接收到正确的设备地址，发送应答脉冲 A(启用 ACK)后，I2C 控制器产生事件 EV1，ADDR1 被置 1，读 I2C_SR1 寄存器然后再读 I2C_SR2 寄存器可清除 ADDR 位。

若此时移位寄存器和数据寄存器中均没有数据，TxE=1，此时产生事件 EV3_1，可将数据写入数据寄存器 DR。

写入数据寄存器 DR 后，TxE=0，此时移位寄存器为空，则数据立即从数据寄存器写入到移位寄存器，移位寄存器非空，TxE=1，产生事件 EV3；如果接收设备地址 R/W 方向控制位为 0，则从发送器将字节从 DR 寄存器经由内部移位寄存器发送到 SDA 线上。

一个数据发送完成后，从发送器接收主设备的应答脉冲(启用 ACK)，若收到应答脉冲，TxE 位被硬件置 1，如果设置了 ITEVFEN 和 ITBUFEN 位，则产生一个中断。如果 TxE 位被置 1，且收到了 NACK(AF=1)，产生事件 EV3_2，则从设备不会再向主设备发送数据，在下一个数据发送结束之前没有新数据写入到 I2C_DR 寄存器，则 BTF 位被置 1，在清除 BTF 之前 I2C 接口将保持 SCL 为低电平；读出 I2C_SR1 之后再写入 I2C_DR 寄存器将清除 BTF 位。

2. 从模式下的接收

从模式下的接收指的是主设备向从设备写数据，从设备不向主设备发送数据。从模式接收的流程如图 9-24 所示。在接收到地址(产生事件 EV1)并清除 ADDR 后，从接收器将通过内部移位寄存器从 SDA 线接收到的字节存进 DR 寄存器。I2C 接口在接收到每个字节后都执行下列操作：

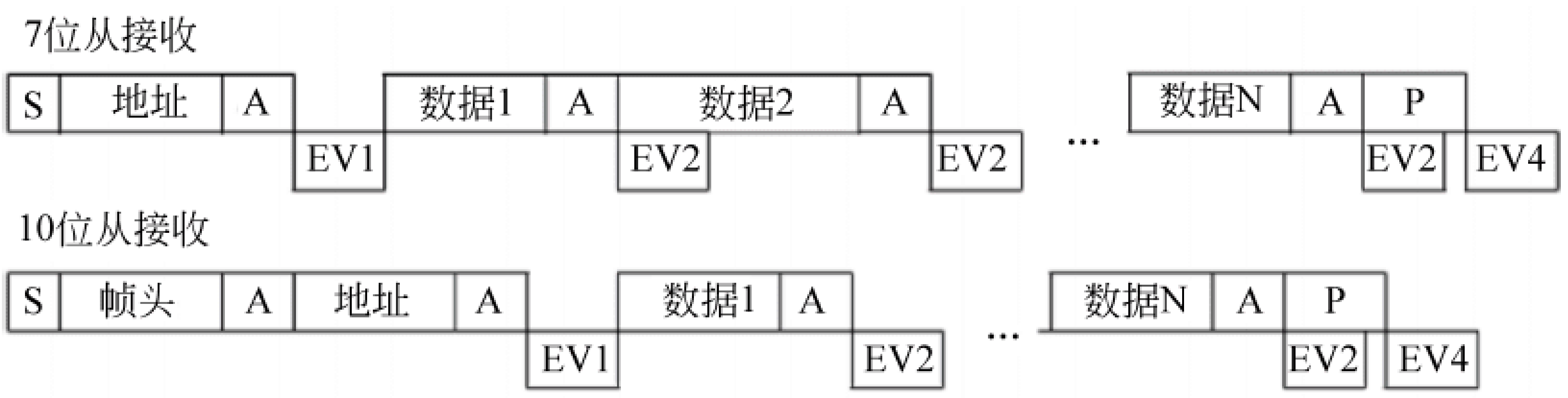


图 9-24 从模式接收时序

- 如果设置了 ACK 位，则产生一个应答脉冲。
- 硬件设置 RxNE=1(产生事件 EV2)，如果设置了 ITEVFEN 和 ITBUFEN 位，则产生一个中断。

如果 RxNE 被置位，并且在接收新的数据结束之前 DR 寄存器未被读出，BTF 位被置位，在清除 BTF 之前 I2C 接口将保持 SCL 为低电平；读出 I2C_SR1 之后再写入 I2C_DR 寄存器将清除 BTF 位。

在传输完最后一个数据字节后，主设备产生一个停止条件，I2C 接口检测到这一条件时：设置 STOPF=1，产生事件 EV4，如果设置了 ITEVFEN 位，则产生一个中断。读 SR1 寄存器，再写 CR1 寄存器可清除 STOPF 位。

9.5.2　I2C 主模式通信

在主模式时，I2C 接口启动数据传输并产生时钟信号。串行数据传输总是以起始条件开始并以停止条件结束。当通过 START 位在总线上产生了起始条件，设备就进入了主模式。以下是主模式的操作顺序：

(1) 在 I2C_CR2 寄存器中设定该模块的输入时钟以产生正确的时序，输入时钟在标准模式下至少为 2MHz，快速模式下至少为 4MHz。

(2) 配置时钟控制寄存器。

(3) 配置上升时间寄存器。

(4) 编程 I2C_CR1 寄存器启动外设。

(5) 置 I2C_CR1 寄存器中的 START 位为 1，产生开始条件。

当 BUSY=0 时，设置 START=1，I2C 接口将产生一个开始条件并切换至主模式(M/SL 位置位)。在主模式下，设置 START=1 将在当前字节传输完后由硬件产生一个重开始条件。

一旦发出开始条件，SB 位被硬件置为 1，产生事件 EV5，如果设置了 ITEVFEN 位，则产生一个中断。然后主设备写 DR 寄存器(从设备地址)清除 SB 位，等待读 SR1 寄存器。

(6) 在 10 位地址模式时，先发送一个头段序列，若收到一个响应 ACK，则 ADD10 位被硬件置位，产生事件 9，如果设置了 ITEVFEN 位，则产生一个中断。读 SR1 寄存器，再将第二个地址字节写入 DR 寄存器(清除 ADD10)，若收到一个响应 ACK，则 ADDR 位被硬件置位，产生事件 EV6，如果设置了 ITEVFEN 位，则产生一个中断。主设备等待一次读 SR1 寄存器和读 SR2 寄存器清除 ADDR 位。

(7) 在 7 位地址模式时，只需送出一个地址字节。一旦该地址字节被送出，收到 ACK 响应，ADDR 位被硬件置为 1，产生事件 EV6，如果设置了 ITEVFEN 位，则产生一个中断。主设备等待一次读 SR1 寄存器和一次读 SR2 寄存器清除 ADDR 位。

根据送出从地址的最低位，主设备决定进入发送器模式还是进入接收器模式。在 7 位地址模式时，要进入发送器模式，主设备发送从地址时置最低位为 0，要进入接收器模式，主设备发送从地址时置最低位为 1。

在 10 位地址模式时，要进入发送器模式，主设备先送头字节 11110xx0(xx 代表 10 位地址中的最高 2 位)，再发剩余的 8 位地址；要进入接收器模式，主设备先送头字节(11110xx0)，再发剩余的 8 位地址，然后再重新发送一个开始条件，后面跟着头字节(11110xx1)。

TRA 位指示主设备是在接收器模式还是发送器模式。

1. 主模式下的发送

主模式下的发送时序如图 9-25 所示，在发送了地址和清除了 ADDR 位后，主设备通过内部移位寄存器将字节从 DR 寄存器发送到 SDA 线上。

若此时数据寄存器和移位寄存器为空，则 TxE=1，产生事件 EV8_1，写入数据到数据

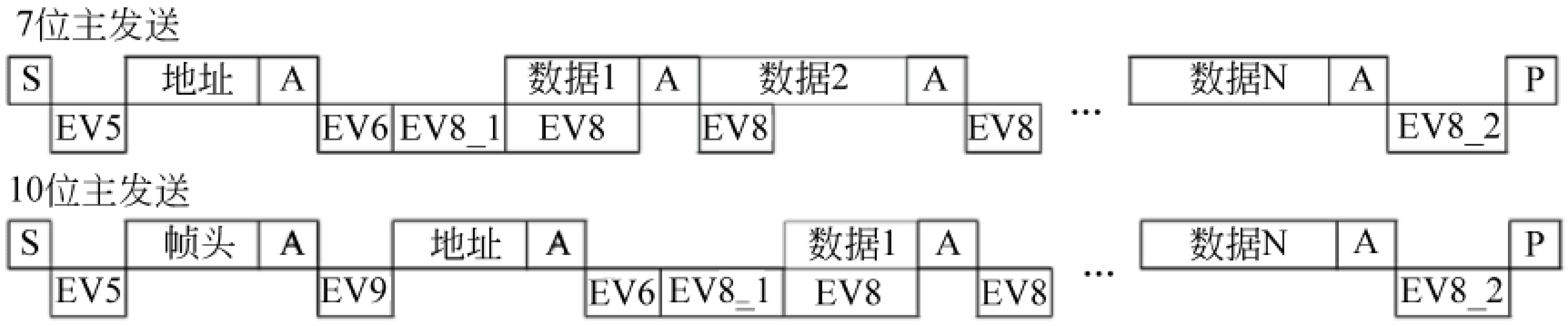


图 9-25　主模式下的发送时序

寄存器 DR 后，TxE＝0，此时由于移位寄存器为空，则数据立即被送到移位寄存器发送，TxE＝1，此时产生事件 EV8。当收到应答脉冲时，TxE 位被硬件置位，产生 EV8，如果 TxE 被置位并且在上一次数据发送结束之前没有写新的数据字节到 DR 寄存器，则 BTF 被硬件置位，产生事件 EV8_2，在清除 BTF 之前 I2C 接口将保持 SCL 为低电平；读出 I2C_SR1 之后再写入 I2C_DR 寄存器将清除 BTF 位。

在 DR 寄存器中写入最后一个字节后，通过设置 STOP 位产生一个停止条件，然后 I2C 接口将自动回到从模式（M/S 位清除）。

2. 主模式下的接收

主接收模式下，主设备向从设备发送起始信号和地址，然后接收从设备发来的数据。如图 9-26 所示，主设备首先发起起始信号，等待事件 EV5，然后发送地址和读命令，从设备返回 ACK 后产生 EV6 事件，清除 ADDR 之后，I2C 接口进入主接收器模式。在此模式下，I2C 接口从 SDA 线接收数据字节，并通过内部移位寄存器送至 DR 寄存器。在每个字节后，如果 ACK 位被置 1，发出一个应答脉冲，硬件设置 RxNE＝1，产生事件 EV7 如果设置了 INEVFEN 和 ITBUFEN 位，则会产生一个中断。如果 RxNE 位被置位，并且在接收新数据结束前，DR 寄存器中的数据没有被读走，硬件将设置 BTF＝1，在清除 BTF 之前 I2C 接口将保持 SCL 为低电平；读出 I2C_SR1 之后再读出 I2C_DR 寄存器将清除 BTF 位。

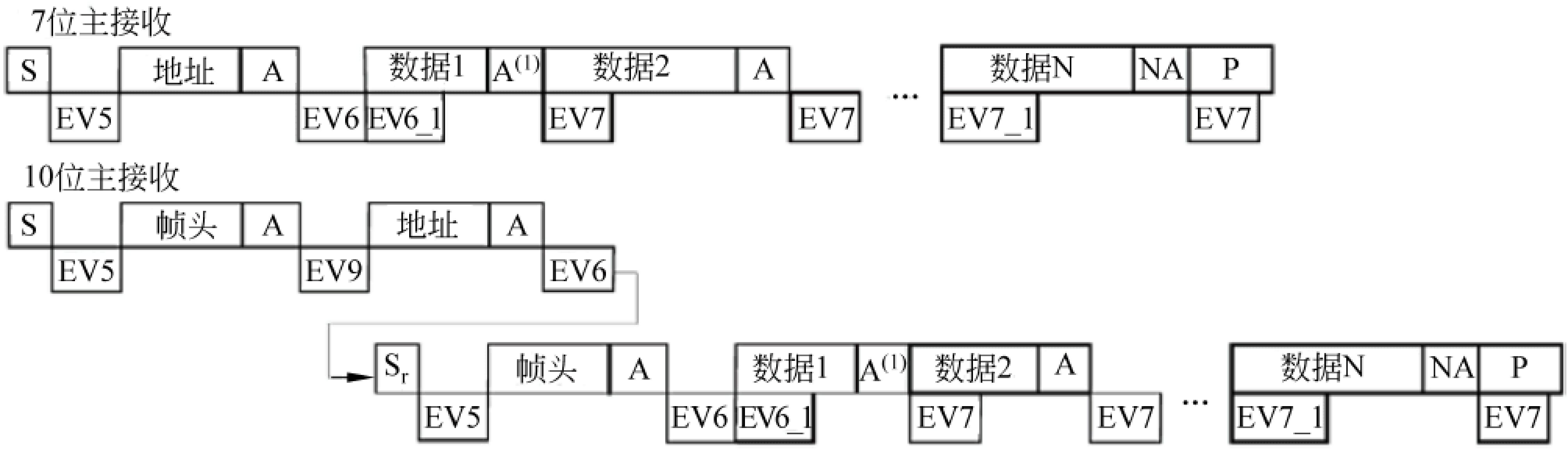


图 9-26　主接收模式时序

主设备在从从设备接收到最后一个字节后发送一个 NACK。接收到 NACK 后，从设备释放对 SCL 和 SDA 线的控制，主设备就可以发送下一个停止或重起始信号。为了在收到最后一个字节后产生一个 NACK 脉冲，在读倒数第二个数据字节之后（在倒数第二个 RxNE 事件之后）必须清除 ACK 位，并设置 STOP 位或者设置 START 位。如果只接收一个字节时，在第一个 EV6 事件要关闭应答和设置停止条件的产生位。在产生了停止条件

后，I2C 接口自动回到从模式(M/SL 位被清除)。

9.5.3　总线通信错误

I2C 总线的通信可能会由于一些原因造成通信失败。

1. 总线错误(BERR)

在一个地址或数据字节传输期间，当 I2C 接口检测到一个外部的停止或起始条件则产生总线错误，BERR 位被置位为 1，如果设置了 ITERREN 位，则产生一个中断。

在主模式情况下，硬件不释放总线，同时不影响当前的传输状态。此时由软件决定是否要中止当前的传输。在从模式情况下，数据被丢弃，硬件释放总线：

(1) 如果是错误的开始条件，从设备认为是一个重启动，并等待地址或停止条件。

(2) 如果是错误的停止条件，从设备按正常的停止条件操作，同时硬件释放总线。

2. 应答错误(AF)

当接口检测到一个无应答位时，产生应答错误，AF 位被置位，如果设置了 ITERREN 位，则产生一个中断。主发送模式收到 NACK 时，需要产生一个停止条件，从发送模式收到 NACK 时，释放总线。

3. 仲裁丢失(ARLO)

多主设备通信情况下，当 I2C 接口检测到仲裁丢失时产生仲裁丢失错误，ARLO 位被硬件置位，如果设置了 ITERREN 位，则产生一个中断，I2C 接口自动回到从模式(M/SL 位被清除)，释放总线。

4. 过载/欠载错误(OVR)

在从模式下，如果禁止时钟延长(NOSTRETCH＝1)，I2C 接口正在接收数据时，当它已经接收到一个字节(RxNE＝1)，但在 DR 寄存器中前一个字节数据还没有被读出，则发生过载错误。此时，最后接收的数据被丢弃；I2C 接口正在发送数据时，在下一个字节的时钟到达之前，新的数据还未写入 DR 寄存器(TxE＝1)，则发生欠载错误。此时，DR 寄存器中的前一个字节将被重复发出。过载和欠载时，用户需要自己控制发送端是否重发因接收端过载丢失的数据，还是接收端丢失因发送端欠载重复发送的数据。

如果允许时钟延长(NOSTRETCH＝0)，发送器模式下，如果 TxE＝1 且 BTF＝1，I2C 接口在传输前保持时钟线为低，以等待软件读取 SR1，然后把数据写进数据寄存器(缓冲器和移位寄存器都是空的)；接收器模式：如果 RxNE＝1 且 BTF＝1，I2C 接口在接收到数据字节后保持时钟线为低，以等待软件读 SR1，然后读数据寄存器 DR(缓冲器和移位寄存器都是满的)。因此允许时钟延长实际上是一种流量控制的方法，可以解决过载和欠载问题。

9.5.4　中断请求

I2C 的中断源如表 9-2 所示。

表 9-2　I2C 中断请求表

中 断 事 件	事 件 标 志	开启控制位
起始位已发送(主)	SB	ITEVFEN
地址已发送(主)或地址匹配(从)	ADDR	
10 位头段已发送(主)	ADD10	
已收到停止(从)	STOPF	
数据字节传输完成	BTF	
接收缓冲区非空	RxNE	ITEVFEN 和 ITBUFEN
发送缓冲区空	TxE	
总线错误	BERR	ITERREN
仲裁丢失(主)	ARLO	
响应失败	AF	
过载/欠载	OVR	
PEC 错误	PECERR	
超时/Tlow 错误	TIMEOUT	
SMBus 提醒	SMBALERT	

三个中断开关 ITEVFEN、ITBUFEN、ITERREN 的关系如图 9-27 所示，如果要使用 TxE 和 RxNE，则需要打开 ITEVFEN 和 ITBUFEN。每个 I2C 控制器连接到 NVIC 的中断线有两个，分别是 IT_EVENT 和 IT_ERROR。

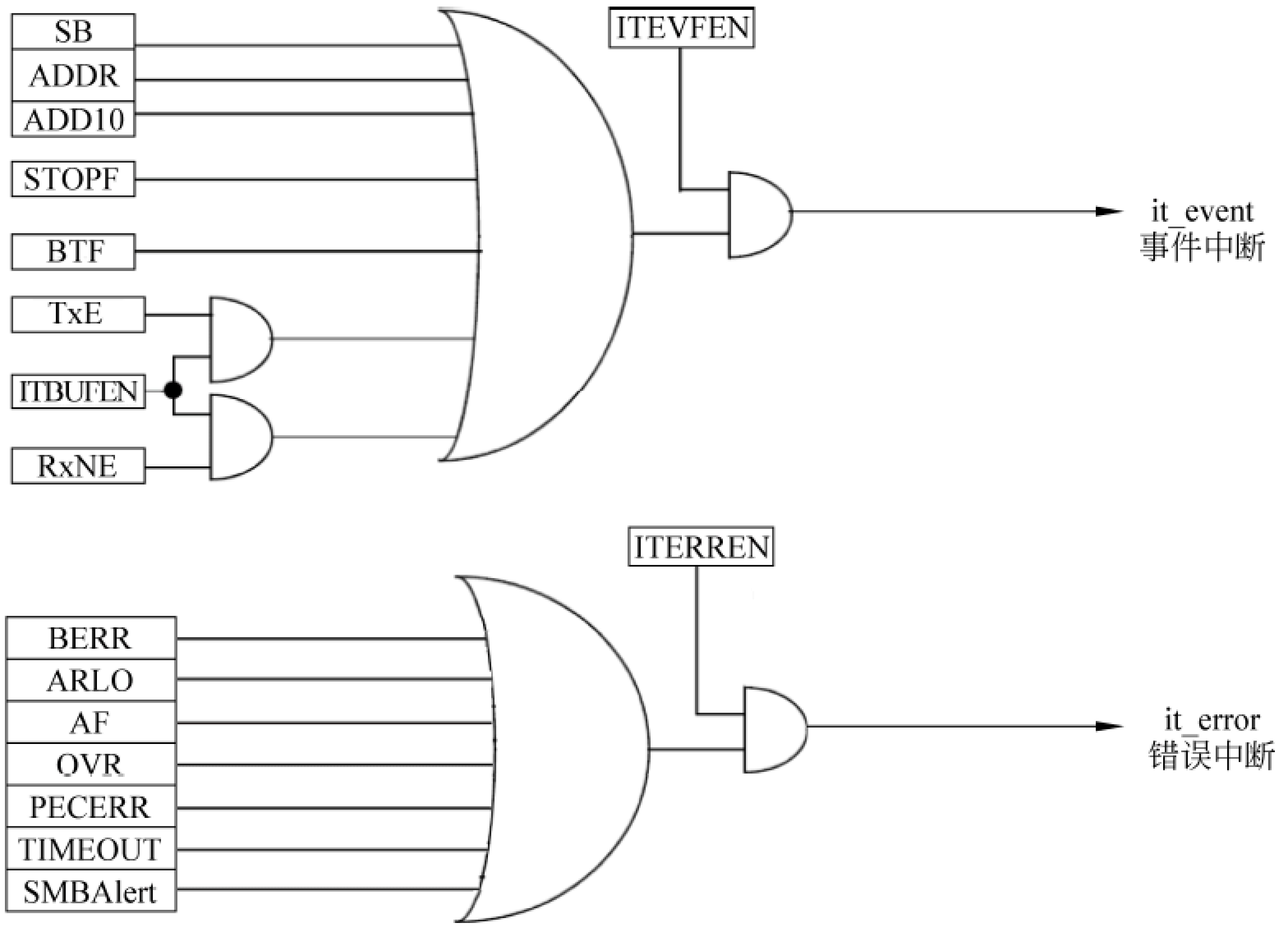


图 9-27　I2C 中断映射

I2C 用到的 I/O 引脚如表 9-3 所示。

表 9-3　IIC 外部引脚

I2C 引脚功能	I2C1 I/O 引脚	I2C2 I/O 引脚
I2C_SCL	PB6、PB8	PB10
I2C_SDA	PB7、PB9	PB11
I2C_SMBA	PB5	PB12

9.6　函数库

9.6.1　I2C 寄存器结构

I2C 寄存器结构，I2C_TypeDeff 在文件 stm32lxxx.h 中定义如下：

```
typedef struct
{
    __IO uint16_t CR1;                    // I2C 控制寄存器 1
    uint16_t RESERVED0;
    __IO uint16_t CR2;                    // I2C 控制寄存器 2
    uint16_t RESERVED1;
    __IO uint16_t OAR1;                   // I2C 自身地址寄存器 1
    uint16_t RESERVED2;
    __IO uint16_t OAR2;                   // I2C 自身地址寄存器 2
    uint16_t RESERVED3;
    __IO uint16_t DR;                     // I2C 数据寄存器
    uint16_t RESERVED4;
    __IO uint16_t SR1;                    // I2C 状态寄存器 1
    uint16_t RESERVED5;
    __IO uint16_t SR2;                    // I2C 状态寄存器 2
    uint16_t RESERVED6;
    __IO uint16_t CCR;                    // I2C 时钟控制寄存器
    uint16_t RESERVED7;
    __IO uint16_t TRISE;                  // I2C 上升时间寄存器
    uint16_t RESERVED8;
} I2C_TypeDef;
```

2 个 I2C 外设声明于文件 stm32l1xx.h 中：

```
#define PERIPH_BASE ((uint32_t)0x40000000)
#define APB1PERIPH_BASE PERIPH_BASE
```

```
#define APB2PERIPH_BASE (PERIPH_BASE + 0x10000)
#define AHBPERIPH_BASE (PERIPH_BASE + 0x20000)
#define I2C1_BASE (APB1PERIPH_BASE + 0x5400)
#define I2C2_BASE (APB1PERIPH_BASE + 0x5800)
#define I2C1 (I2C_TypeDef * ) I2C1_BASE
#define I2C2 (I2C_TypeDef * ) I2C2_BASE
```

用于 I2C 寄存器初始化的 I2C_InitTypeDef 结构体定义于文件 stm32l1xx_i2c. h：

```
typedef struct
{
  Uint32_t I2C_ClockSpeed;
  uint16_t I2C_Mode;
  uint16_t I2C_DutyCycle;
  uint16_t I2C_OwnAddress1;
  uint16_t I2C_Ack;
  uint16_t I2C_AcknowledgedAddress;
} I2C_InitTypeDef;
```

其中 I2C_ClockSpeed 参数用来设置时钟频率，这个值不能高于 400kHz。

I2C_Mode 用于设置 I2C 的模式，其取值范围为：

- I2C_Mode_I2C　　设置 I2C 为 I2C 模式
- I2C_Mode_SMBusDevice　　设置 I2C 为 SMBus 设备模式
- I2C_Mode_SMBusHost　　设置 I2C 为 SMBus 主控模式

I2C_DutyCycle 用以设置 I2C 的占空比，该参数只有在 I2C 工作在快速模式(时钟工作频率高于 100kHz)下才有意义，其取值范围为：

- I2C_DutyCycle_16_9　　I2C 快速模式 Tlow/Thigh=16/9
- I2C_DutyCycle_2　　I2C 快速模式 Tlow/Thigh=2

I2C_OwnAddress1 该参数用来设置第一个设备自身地址，它可以是一个 7 位地址或者一个 10 位地址。

I2C_Ack 使能或者失能应答(ACK)，其取值为：

- I2C_Ack_Enable　　使能应答(ACK)
- I2C_Ack_Disable　　失能应答(ACK)

I2C_AcknowledgedAddres 定义了应答 7 位地址还是 10 位地址，其取值范围为：

- I2C_AcknowledgeAddress_7bit　　应答 7 位地址
- I2C_AcknowledgeAddress_10bit　　应答 10 位地址

9.6.2　I2C 库函数

ST 提供的 I2C 标准函数库如表 9-4 所示。

表 9-4　I2C 函数列表

函　数　名	描　　述
I2C_DeInit	将外设 I2Cx 寄存器重设为默认值
I2C_Init	根据 I2C_InitStruct 中指定的参数初始化外设 I2Cx 寄存器
I2C_StructInit	把 I2C_InitStruct 中的每一个参数按默认值填入
I2C_Cmd	使能或者失能 I2C 外设
I2C_GenerateSTART	产生 I2Cx 传输 START 条件
I2C_GenerateSTOP	产生 I2Cx 传输 STOP 条件
I2C_AcknowledgeConfig	使能或者失能指定 I2C 的应答功能
I2C_OwnAddress2Config	设置指定 I2C 的自身地址 2
I2C_DualAddressCmd	使能或者失能指定 I2C 的双地址模式
I2C_GeneralCallCmd	使能或者失能指定 I2C 的广播呼叫功能
I2C_SoftwareResetCmd	使能或者失能指定 I2C 的软件复位
I2C_SMBusAlertConfig	驱动指定 I2Cx 的 SMBusAlert 引脚电平为高或低
I2C_ARPCmd	使能或者失能指定 I2C 的 ARP
I2C_StretchClockCmd	使能或者失能指定 I2C 的时钟延展
I2C_FastModeDutyCycleConfig	选择指定 I2C 的快速模式占空比
I2C_Send7bitAddress	向指定的从 I2C 设备传送地址字
I2C_SendData	通过外设 I2Cx 发送一个数据
I2C_ReceiveData	返回通过 I2Cx 最近接收的数据
I2C_NACKPositionConfig	2 字节接收时 NACK 的发送位置
I2C_TransmitPEC	使能或者失能指定 I2C 的 PEC 传输
I2C_PECPositionConfig	选择指定 I2C 的 PEC 位置
I2C_CalculatePEC	使能或者失能指定 I2C 的传输字 PEC 值计算
I2C_GetPEC	返回指定 I2C 的 PEC 值
I2C_DMACmd	使能或者失能指定 I2C 的 DMA 请求
I2C_DMALastTransferCmd	使下一次 DMA 传输为最后一次传输
I2C_ReadRegister	读取指定的 I2C 寄存器并返回其值
I2C_ITConfig	使能或者失能指定的 I2C 中断
I2C_CheckEvent	检查最近一次 I2C 事件是否是输入的事件
I2C_GetLastEvent	返回最近一次 I2C 事件
I2C_GetFlagStatus	检查指定的 I2C 标志位设置与否

续表

函　数　名	描　　述
I2C_ClearFlag	清除 I2Cx 的待处理标志位
I2C_GetITStatus	检查指定的 I2C 中断发生与否
I2C_ClearITPendingBit	清除 I2Cx 的中断待处理位

在调用 I2C 库函数前，需要打开 I2C 总线时钟，调用 RCC_APB1PeriphClockCmd()。

1）函数 I2C_DeInit

功能描述：将外设 I2Cx 寄存器重设为默认值。

函数原型：void I2C_DeInit(I2C_TypeDef * I2Cx)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

示例：

```
I2C_DeInit(I2C2);
```

2）函数 I2C_Init

功能描述：根据 I2C_InitStruct 中指定的参数初始化外设 I2Cx 寄存器。

函数原型：void I2C_Init(I2C_TypeDef * I2Cx，I2C_InitTypeDef * I2C_InitStruct)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 I2C_InitStruct：指向结构 I2C_InitTypeDef 的指针。

示例：

```
I2C_InitTypeDef I2C_InitStructure;
I2C_InitStructure.I2C_Mode = I2C_Mode_SMBusHost;
I2C_InitStructure.I2C_DutyCycle = I2C_DutyCycle_2;
I2C_InitStructure.I2C_OwnAddress1 = 0x03A2;
I2C_InitStructure.I2C_Ack = I2C_Ack_Enable;
I2C_InitStructure.I2C_AcknowledgedAddress =
I2C_AcknowledgedAddress_10bit;
I2C_InitStructure.I2C_ClockSpeed = 200000;
I2C_Init(I2C1, &I2C_InitStructure);
```

3）函数 I2C_StructInit

功能描述：把 I2C_InitStruct 中的每一个参数按默认值填入。

函数原型：void I2C_StructInit(I2C_InitTypeDef * I2C_InitStruct)。

输入参数 I2C_InitStruct，指向结构 I2C_InitTypeDef 的指针，待初始化。

I2C_InitStruct 各个成员的默认值为：

- I2C_Mode　　I2C_Mode_I2C
- I2C_DutyCycle　　I2C_DutyCycle_2
- I2C_OwnAddress1　　0

- I2C_Ack　I2C_Ack_Disable
- I2C_AcknowledgedAddres　I2C_AcknowledgedAddress_7bit
- I2C_ ClockSpeed　5000

示例：

```
I2C_InitTypeDef I2C_InitStructure;
I2C_StructInit(&I2C_InitStructure);
```

4）函数 I2C_Cmd

功能描述：使能或者禁用 I2C 外设。

函数原型：void I2C_Cmd(I2C_TypeDef * I2Cx，FunctionalState NewState)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 NewState：外设 I2Cx 的状态，取值为 ENABLE 或者 DISABLE。

示例：

```
I2C_Cmd(I2C1,ENABLE);
```

5）函数 I2C_GenerateSTART

功能描述：产生 I2Cx 的 START 信号。

函数原型：void I2C_GenerateSTART(I2C_TypeDef * I2Cx，FunctionalState NewState)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 NewState，I2Cx START 条件的状态，取值为 ENABLE 或者 DISABLE。

示例：

```
I2C_GenerateSTART(I2C1,ENABLE);
```

6）函数 I2C_GenerateSTOP

功能描述：产生 I2Cx 的 STOP 信号。

函数原型：void I2C_GenerateSTOP(I2C_TypeDef * I2Cx，FunctionalState NewState)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 NewState：表示 I2Cx STOP 条件的新状态，取值为 ENABLE 或者 DISABLE。

示例：

```
I2C_GenerateSTOP(I2C2,ENABLE);
```

7）函数 I2C_AcknowledgeConfig

功能描述：使能或者失能指定 I2C 控制器的应答功能。

函数原型：void I2C_AcknowledgeConfig(I2C_TypeDef * I2Cx，FunctionalState NewState)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 NewState：表示 I2Cx STOP 条件的状态，取值为 ENABLE 或者 DISABLE。

示例：

```
I2C_AcknowledgeConfig(I2C1,ENABLE);
```

8）I2C_OwnAddress2Config 函数

功能描述：配置 I2C 自身地址 2。

函数原型：void I2C_OwnAddress2Config(I2C_TypeDef * I2Cx, uint8_t Address)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 Address：I2C 的 7 位地址自身地址 2 的值。

示例：

```
I2C_OwnAddress2Config(I2C1,0x31);
```

9）I2C_DualAddressCmd 函数

功能描述：使能或禁用 I2C 的双地址模式。

函数原型：void I2C_DualAddressCmd(I2C_TypeDef * I2Cx, FunctionalState NewState)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 NewState，表示 I2Cx 双地址模式的状态，取值为 ENABLE 或者 DISABLE。

示例：

```
I2C_DualAddressCmd(I2C1,ENABLE);
```

10）函数 I2C_SoftwareResetCmd

功能描述：使能或者禁用指定 I2C 的软件复位。

函数原型：I2C_SoftwareResetCmd(I2C_TypeDef * I2Cx, FunctionalState NewState)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 NewState 为软件复位的新状态，取值为 ENABLE 或者 DISABLE。

示例：

```
I2C_SoftwareResetCmd(I2C1,ENABLE);
```

11）函数 I2C_StretchClockCmd

功能描述：使能或者禁用指定 I2C 的时钟延长功能。

函数原型：void I2C_StretchClockCmd(I2C_TypeDef * I2Cx,FunctionalState NewState)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 NewState 为时钟延长功能的状态，取值为 ENABLE 或者 DISABLE。

示例：

```
I2C_StretchClockCmd(I2C1,ENABLE);
```

12）函数 I2C_FastModeDutyCycleConfig

功能描述：选择指定 I2C 的快速模式占空比。

函数原型：void I2C_FastModeDutyCycleConfig(I2C_TypeDef * I2Cx, uint16_t I2C_DutyCycle)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 I2C_DutyCycle 用来指定快速模式占空比，其取值为：

- I2C_DutyCycle_16_9　I2C 快速模式 Tlow/Thigh = 16/9
- I2C_DutyCycle_2　　I2C 快速模式 Tlow/Thigh = 2

示例：

```
I2C_FastModeDutyCycleConfig(I2C2, I2C_DutyCycle_16_9);
```

13）函数 I2C_Send7bitAddress

功能描述：向指定的从 I2C 设备传送地址字。

函数原型：void I2C_Send7bitAddress(I2C_TypeDef * I2Cx，uint8_t Address，uint8_t I2C_Direction)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 Address 为待传输的从 I2C 设备地址。

输入参数 I2C_Direction 用于设置指定的 I2C 设备工作为发送模式还是接收模式，其取值为：

- I2C_Direction_Transmitter　选择发送模式
- I2C_Direction_Receiver　　选择接收模式

示例：

```
I2C_Send7bitAddress(I2C1, 0xA8, I2C_Direction_Transmitter);
```

14）函数 I2C_SendData

功能描述：通过外设 I2Cx 发送一个数据。

函数原型：void I2C_SendData(I2C_TypeDef * I2Cx，uint8_t Data)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 Data 为待发送的数据。

示例：

```
I2C_SendData(I2C2,0x5D);
```

15）函数 I2C_ ReceiveData

功能描述：返回通过 I2Cx 最近接收的数据。

函数原型：uint8_t I2C_ReceiveData(I2C_TypeDef * I2Cx)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

示例：

```
uint8_t ReceivedData;
ReceivedData = I2C_ReceiveData(I2C1);
```

16）函数 I2C_NACKPositionConfig

功能描述：在主模式接收时，两个字节读取下确认 ACK 的发送时机。

函数原型：uint8_t I2C_ReceiveData(I2C_TypeDef * I2Cx)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 I2C_NACKPosition，指定 NACK 的发送位置，其取值为：

- I2C_NACKPosition_Next 下一个字节接收后发 NACK
- I2C_NACKPosition_Current 当前字节接收后发 NACK

示例：

```
I2C_NACKPositionConfig(I2C1,I2C_NACKPosition_Next);
```

17）函数 I2C_ ReadRegister

功能描述：读取指定的 I2C 寄存器并返回其值。

函数原型：uint16_t I2C_ReadRegister(I2C_TypeDef * I2Cx, uint8_t I2C_Register)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 I2C_Register 指定待读取的 I2C 寄存器，其取值为：I2C_Register_CR1、I2C_Register_CR2、I2C_Register_OAR1、I2C_Register_OAR2、I2C_Register_DR、I2C_Register_SR1、I2C_Register_SR2、I2C_Register_CCR、I2C_Register_TRISE。

返回值为被读取的寄存器值。

示例：

```
uint16_t RegisterValue;
RegisterValue = I2C_ReadRegister(I2C2, I2C_Register_CR1);
```

18）函数 I2C_ITConfig

功能描述：使能或者禁用指定的 I2C 中断。

函数原型：void I2C_ITConfig(I2C_TypeDef * I2Cx,uint16_t I2C_IT,FunctionalState NewState)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 I2C_IT 为待使能或者禁用的 I2C 中断源，可以取一个或者多个取值的组合作为该参数的值。

- I2C_IT_BUF 缓存中断屏蔽。
- I2C_IT_EVT 事件中断屏蔽。
- I2C_IT_ERR 错误中断屏蔽。

输入参数 NewState 表示 I2Cx 中断的新状态，取值为 ENABLE 或 DISABLE。

示例：

```
I2C_ITConfig(I2C2, I2C_IT_BUF | I2C_IT_EVT,ENABLE);
```

19）函数 I2C_GetLastEvent

功能描述：返回最近一次 I2C 事件。

函数原型：uint32_t I2C_GetLastEvent(I2C_TypeDef * I2Cx)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

返回值为最近一次 I2C 事件,事件定义见表 9-5。

示例:

```
uint32_t Event;
Event = I2C_GetLastEvent(I2C1);
```

表 9-5　I2C 事件定义

I2C_Event	描　述
I2C_EVENT_SLA VE_RECEIVER_ADDRESS_MA TCHED	EV1
I2C_EVENT_SLAVE_TRANSMITTER_ADDRESS_MATCHED	EV1
I2C_EVENT_SLAVE_RECEIVER_SECONDADDRESS_MATCHED	EV1
I2C_EVENT_SLAVE_TRANSMITTER_SECONDADDRESS_MA TCHED	EV1
I2C_EVENT_SLAVE_GENERALCALLADDRESS_MATCHED	EV1
I2C_EVENT_SLA VE_BYTE_RECEIVED	EV2
I2C_EVENT_SLAVE_BYTE_TRANSMITTED	EV3
I2C_EVENT_SLAVE_ACK_FAILURE	EV3～1
I2C_EVENT_SLA VE_STOP_DETECTED	EV4
I2C_EVENT_MASTER_MODE_SELECT	EV5
I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED	EV6
I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED	EV6
I2C_EVENT_MASTER_BYTE_RECEIVED	EV7
I2C_EVENT_MASTER_BYTE_TRANSMITTED	EV8
I2C_EVENT_MASTER_MODE_ADDRESS10	EV9

20) 函数 I2C_CheckEvent

功能描述:检查最近一次 I2C 事件是否是输入的事件。

函数原型:ErrorStatus I2C_CheckEvent(I2C_TypeDef * I2Cx, uint32_t I2C_EVENT)。

输入参数 I2Cx:用来选择 I2C 外设,x 可以是 1 或者 2。

输入参数 I2C_Event 用于指定待检查的事件,其取值见表 9-5。

返回值 ErrorStatus 取值为 SUCCESS 或 ERROR,如果是 SUCCESS 表明所检查的事件是最近一次 I2C 事件,ERROR 表明最近一次 I2C 事件不是所检查的事件。

示例:

```
ErrorStatus Status;
Status = I2C_CheckEvent(I2C1,I2C_EVENT_MSTER_BYTE_RECEIVED);
```

21）函数 I2C_GetFlagStatus

功能描述：检查指定的 I2C 标志位设置与否。

函数原型：FlagStatus I2C_GetFlagStatus(I2C_TypeDef * I2Cx，uint32_t I2C_FLAG)。

输入参数 I2Cx 用来选择 I2C 外设，I2C_FLAG 用于指定待检查的 I2C 标志位，其取值范围如表 9-6 所示。

返回值为 I2C_FLAG 的状态，取值范围为 SET 或 RESET。

表 9-6 I2C_FLAG 的取值

I2C_FLAG	描 述
I2C_FLAG_DUALF	双标志位(从模式)
I2C_FLAG_SMBHOST	SMBus 主报头(从模式)
I2C_FLAG_SMBDEFAULT	SMBus 缺省报头(从模式)
I2C_FLAG_GENCALL	广播报头标志位(从模式)
I2C_FLAG_TRA	发送/接收标志位
I2C_FLAG_BUSY	总线忙标志位
I2C_FLAG_MSL	主/从标志位
I2C_FLAG_SMBALERT	SMBus 报警标志位
I2C_FLAG_TIMEOUT	超时或者 Tlow 错误标志位
I2C_FLAG_PECERR	接收 PEC 错误标志位
I2C_FLAG_OVR	溢出/不足标志位(从模式)
I2C_FLAG_AF	应答错误标志位
I2C_FLAG_ARLO	仲裁丢失标志位(主模式)
I2C_FLAG_BERR	总线错误标志位
I2C_FLAG_TXE	数据寄存器空标志位(发送端)
I2C_FLAG_RXNE	数据寄存器非空标志位(接收端)
I2C_FLAG_STOPF	停止标志位(从模式)
I2C_FLAG_ADD10	10 位报头发送(主模式)
I2C_FLAG_BTF	字传输完成标志位
I2C_FLAG_ADDR	地址发送标志位(主模式)或地址匹配标志位(从模式)ADDR
I2C_FLAG_SB	起始位标志位(主模式)

示例：

```
Flagstatus Status;
Status = I2C_GetFlagStatus(I2C2,I2C_FLAG_AF);
```

22）函数 I2C_ClearFlag

功能描述：清除 I2Cx 的待处理标志位。

函数原型：void I2C_ClearFlag(I2C_TypeDef * I2Cx, uint32_t I2C_FLAG)。

输入参数 I2Cx 用来选择 I2C 外设，I2C_FLAG 为待清除的 I2C 标志位，取值见表 9-6。但 DUALF，SMBHOST，SMBDEFAULT，GENCALL，TRA，BUSY，MSL，TXE 和 RXNE 不能被本函数清除。

示例：

```
I2C_ClearFlag(I2C2, I2C_FLAG_STOPF);
```

23）函数 I2C_GetITStatus

功能描述：检查指定的 I2C 中断发生与否。

函数原型：ITStatus I2C_GetITStatus(I2C_TypeDef * I2Cx, uint32_t I2C_IT)。

输入参数 I2Cx 用来选择 I2C 外设，I2C_IT 为待检查的 I2C 中断源，其取值见表 9-7。

返回值为 I2C_IT 的状态，取值为 SET 或者 RESET。

表 9-7　I2C_IT 值

I2C_IT	描　　述
I2C_IT_SMBALERT	SMBus 报警标志位
I2C_IT_TIMEOUT	超时或者 Tlow 错误标志位
I2C_IT_PECERR	接收 PEC 错误标志位
I2C_IT_OVR	溢出/不足标志位(从模式)
I2C_IT_AF	应答错误标志位
I2C_IT_ARLO	仲裁丢失标志位(主模式)
I2C_IT_BERR	总线错误标志位
I2C_IT_STOPF	停止探测标志位(从模式)
I2C_IT_ADD10	10 位报头发送(主模式)
I2C_IT_BTF	字传输完成标志位
I2C_IT_ADDR	地址发送标志位(主模式)与地址匹配标志位(从模式)ADDR
I2C_IT_SB	起始位标志位(主模式)

示例：

```
ITstatus Status;
Status = I2C_GetITStatus(I2C1, I2C_IT_OVR);
```

24）函数 I2C_ClearITPendingBit

功能描述：清除 I2Cx 的中断待处理位

函数原型：void I2C_ClearITPendingBit(I2C_TypeDef * I2Cx, uint32_t I2C_IT)。

输入参数 I2Cx 用来选择 I2C 外设,I2C_IT 指定待清除的 I2C 中断源,取值见表 9-6。
示例:

```
I2C_ClearITPendingBit(I2C2, I2C_IT_TIMEOUT);
```

9.7 I2C 案例

9.7.1 I2C 寄存器操作案例

1) I2C_Init 函数的实现

```
void I2C_Init(I2C_TypeDef * I2Cx, I2C_InitTypeDef * I2C_InitStruct)
{
  uint16_t tmpreg = 0, freqrange = 0;
  uint16_t result = 0x04;
  uint32_t pclk1 = 8000000;
  RCC_ClocksTypeDef  rcc_clocks;
  // CR2 寄存器配置
  tmpreg = I2Cx->CR2;                    //获取 CR2 的值
  // 清除频率字段 FREQ[5:0]
  tmpreg &= (uint16_t)~ ((uint16_t)I2C_CR2_FREQ);
  // 配置频率字段,首选获取 APB1 总线时钟
  RCC_GetClocksFreq(&rcc_clocks);
  pclk1 = rcc_clocks.PCLK1_Frequency;
  //将时钟值除以 1M,得到一个整数,作为 FREQ[5:0]的配置,实际频率即为总线频率
  freqrange = (uint16_t) (pclk1 / 1000000);
  tmpreg |= freqrange;
  //写会到 CR2 寄存器中,频率配置完成
  I2Cx->CR2 = tmpreg;
  // CCR 寄存器配置,配置前首先禁用 I2C
  I2Cx->CR1 &= (uint16_t)~ ((uint16_t)I2C_CR1_PE);
  //将 tmpreg 清 0,即 F/S, DUTY 和 CCR[11:0] 均置为 0
  tmpreg = 0;
  //标准模式下的配置
  if (I2C_InitStruct->I2C_ClockSpeed <= 100000)
  {
    //通过时钟频率计算 CCR 的值,最小值为 4
    result = (uint16_t) (pclk1 / (I2C_InitStruct->I2C_ClockSpeed << 1));
    if (result < 0x04)
      result = 0x04;
    tmpreg |= result;        //写入 CCR
```

```
    //配置上升沿时间为 I2C 的 FREQ 字段值+1,即最大上升时间为:1000ns
    I2Cx->TRISE = freqrange + 1;
  }
  //如果是快速模式, PCLK1 必须是 10 MHz 的整数倍 (FREQ 字段配置)
  else                        //(I2C_InitStruct->I2C_ClockSpeed <= 400000)
  { //快速模式下需要配置占空比,不同占空比计算 CCR 的值不一样
    if (I2C_InitStruct->I2C_DutyCycle == I2C_DutyCycle_2)
    {
      result = (uint16_t) (pclk1 / (I2C_InitStruct->I2C_ClockSpeed * 3));
    }
    else                      //I2C_InitStruct->I2C_DutyCycle == I2C_DutyCycle_16_9
    {
        result = (uint16_t) (pclk1 / (I2C_InitStruct->I2C_ClockSpeed * 25));
        result |= I2C_DutyCycle_16_9;
    }
    //CCR 最小值不能小于 1
    if ((result & I2C_CCR_CCR) == 0)
      result |= (uint16_t)0x0001;
    //设置快速模式位和最大上升时延
    tmpreg |= (uint16_t) (result | I2C_CCR_FS);
    I2Cx->TRISE = (uint16_t) (((freqrange * (uint16_t)300) / (uint16_t)1000) + (uint16_t)1);
  }
  //写入 CCR,使能 I2C
  I2Cx->CCR = tmpreg;
  I2Cx->CR1 |= I2C_CR1_PE;
  //CR1 寄存器配置
  tmpreg = I2Cx->CR1;
  //清除 ACK, SMBTYPE 和 SMBUS 位
  tmpreg &= CR1_CLEAR_MASK;
  //根据 I2C_Mode value 和 I2C_Ack value 配置 CR1 寄存器
  tmpreg |= (uint16_t) ((uint32_t)I2C_InitStruct->I2C_Mode
            | I2C_InitStruct->I2C_Ack);
  I2Cx->CR1 = tmpreg;
  // OAR1 寄存器配置
  I2Cx->OAR1 = (I2C_InitStruct->I2C_AcknowledgedAddress
                | I2C_InitStruct->I2C_OwnAddress1);
}
```

9.7.2　I2C 基本配置

1）基本配置流程

- 使能时钟 RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2Cx，ENABLE)；
- 使能 SDA，SCL 所使用的 GPIO 端口时钟 RCC_AHBPeriphClockCmd()；

- 使用 GPIO_PinAFConfig()将 GPIO 和 I2C 复用功能进行映射；
- 配置 GPIO：复选功能，输出模式，类型为开漏，调用 GPIO_Init()初始化；
- 通过配置模式，占空比、地址、ACK 等参数，调用 I2C_Init()初始化；
- 如有必要可以单独调用 I2C_AcknowledgeConfig()、I2C_DualAddressCmd()、I2C_FastModeDutyCycleConfig()等函数进行配置；
- 如需中断，配置 NVIC 相应的中断源，并调用 I2C_ITConfig()开启所需的中断；
- 如需 DMA，需要调用 DMA_Init()、I2C_DMACmd()或 I2C_DMALastTransferCmd()；
- 调用 I2C_Cmd()使能 I2C。

2）基本配置案例

使用库函数实现 I2C 的配置和读写时序的三个函数如下：

```
void RCC_Configuration(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    I2C_InitTypeDef I2C_InitStructure;
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOB,ENABLE);
    RCC_AHB1PeriphClock(RCC_APB1Periph_I2C1,ENABLE);
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7 | GPIO_Pin_6;
    GPIO_InitStructure.GPIO_Speed= GPIO_Speed_40MHz;
    GPIO_InitStructure.GPIO_Mode= GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_OType= GPIO_OType_OD;
    GPIO_Init(GPIOB,&GPIO_InitStructure);
    GPIO_PinAFConfig(GPIOB,GPIO_PinSource7,GPIO_AF_I2C1);
    GPIO_PinAFConfig(GPIOB,GPIO_PinSource6,GPIO_AF_I2C1);
    I2C_InitStructure.I2C_Mode= I2C_Mode_I2C;
    I2C_InitStructure.I2C_DutyCycle= I2C_DutyCycle_2;
    I2C_InitStructure.OwnAddress1 = 0xA3;
    I2C_InitStructure.I2C_Ack= I2C_Ack_Enable;
    I2C_InitStructure.I2C_AcknowledgedAddress= I2C_AcknowledgedAddress_7bit;
    I2C_InitStructure.I2C_ClockSpeed = 200000;
    I2C_Init(I2C1,&I2C_InitStructure);
    I2C_Cmd(I2C1,ENABLE);
}
unsigned char I2C_ReadByte(unsigned char Address)
{   //等待 I2C 不忙
    while(I2C_GetFlagStatus(I2C1, I2C_FLAG_BUSY));
    I2C_GenerateSTART(I2C1, ENABLE);                                //重新发送
    while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));     //EV5
    I2C_Send7bitAddress(I2C1, Address, I2C_Direction_Receiver);     //发送地址
    //EV6
    while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED));
    //关闭应答和停止条件产生
```

```
    I2C_AcknowledgeConfig(I2C1, DISABLE);
    I2C_GenerateSTOP(I2C1, ENABLE);
    //等待 EV7
    while(!(I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_RECEIVED)));
    return = I2C_ReceiveData(I2C1);
}
void I2C_WriteByte(unsigned char buff, unsigned char Address)
{
    //产生起始条件
    I2C_GenerateSTART(I2C1,ENABLE);
    while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));
    //向设备发送设备地址
    I2C_Send7bitAddress(I2C1,Address,I2C_Direction_Transmitter);
    //等待 ACK
  while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED));
  I2C_SendData(I2C1, buff);
  //发送完成
  while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));
  //产生结束信号
   I2C_GenerateSTOP(I2C1, ENABLE);
}
```

9.7.3　模拟 I2C 实现

在很多情况下，我们经常使用 GPIO 来模拟 I2C 的时序，这样的方式使得程序的移植更为方便。在 I/O 模拟时，I2C 的总线无需一定要上拉电阻，可以将 I/O 配置为推挽模式。建议配置成开漏，配置上拉电阻，这样无需对端口进行输入输出切换。

```
#define I2C_SLAVE_ADDRESS7 0xA6
#define I2C_SCL_0 GPIO_ResetBits(GPIOB,GPIO_Pin_10)
#define I2C_SCL_1 GPIO_SetBits(GPIOB,GPIO_Pin_10)
#define I2C_SDA_0 GPIO_ResetBits(GPIOB,GPIO_Pin_11)
#define I2C_SDA_1 GPIO_SetBits(GPIOB,GPIO_Pin_11)
#define I2C_SDA_STAT GPIO_ReadInputDataBit(GPIOB,GPIO_Pin_11)
#define I2C_ACK 0
#define I2C_NACK 1
#define I2C_READY 0
#define I2C_BUSY 1
#define I2C_ERROR 3
void NOP(void)
{
    uint8_t i = 5;
```

```
    while(i--);
}
void TWI_Initialize(void)                          //没有上拉电阻将 SDA 和 SCL 设置成推挽输出
{
    GPIO_InitTypeDef GPIO_InitStructure;
    GPIO_InitStructure.GPIO_Speed=GPIO_Speed_10MHz;
    GPIO_InitStructure.GPIO_OType=GPIO_OType_PP;
    GPIO_InitStructure.GPIO_Pin=GPIO_Pin_10 | GPIO_Pin_11;
    GPIO_Init(GPIOB,&GPIO_InitStructure);
}
uint8_t I2C_Start(void)
{
    I2C_SDA_1;
    NOP();
    I2C_SCL_1;
    NOP();
    I2C_SDA_0;
    NOP();
    I2C_SCL_0;
    NOP();
    return I2C_READY;
}
void I2C_STOP(void)
{
    I2C_SDA_0;
    NOP();
    I2C_SCL_1;
    NOP();
    I2C_SDA_1;
    NOP();
}
uint8_t I2C_SendByte(uint8_t data)
{
    uint8_t i,err;
    I2C_SCL_0;
    for(i=0;i<8;i++)
    {
        if(data&0x80)
            I2C_SDA_1;
        else
            I2C_SDA_0;
        data<<=1;
        NOP();                                     //产生一个上升沿
```

```
        I2C_SCL_1;
        NOP();
        I2C_SCL_0;
        NOP();
    }
    I2C_SDA_1;                                    //接收从机应答
    NOP();
    I2C_SCL_1;
    NOP();
    while(I2C_SDA_STAT)
    {
        err++;
        if(err>250)
        {
            I2C_SCL_0;
            I2C_SDA_1;
            return I2C_NACK;
        }
    }
    I2C_SCL_0;
    I2C_SDA_1;
    return I2C_ACK;
}
uint8_t I2C_RecieveByte(void)
{
    uint8_t i,data;
    I2C_SDA_1;
    I2C_SCL_0;
    data=0;
    for(i=0;i<8;i++)
    {
        I2C_SCL_1;
        NOP();
        data<<=1;
        if(I2C_SDA_STAT)
           data|=0x01;
        I2C_SCL_0;
        NOP();
    }
    return data;
}
```

9.7.4 串行 Flash 通信

I2C 的基本时序能够实现两个设备之间的通信，但在实际应用中，我们通常用 MCU 作为主设备去和一些从设备进行数据交换，这些从设备的操作通常需要传输多个字节，例如图 9-28 所示的 SPI 接口串行 Flash AT24C02，存储大小为 256 字节共计 2kb，每个字节均可以随机进行读写访问，此时对于该设备的访问，我们需要两个地址，一个为 I2C 设备地址，用于在 I2C 总线上对 AT24C02 进行寻址，AT24C02 的设备地址的高四位为 1010，低三位由芯片外围的电路连接决定（A2、A1、A0 的电平）；另一个地址为 Flash 的存储地址，即要读写 AT24C02 的 256 个单元的某一个单元。

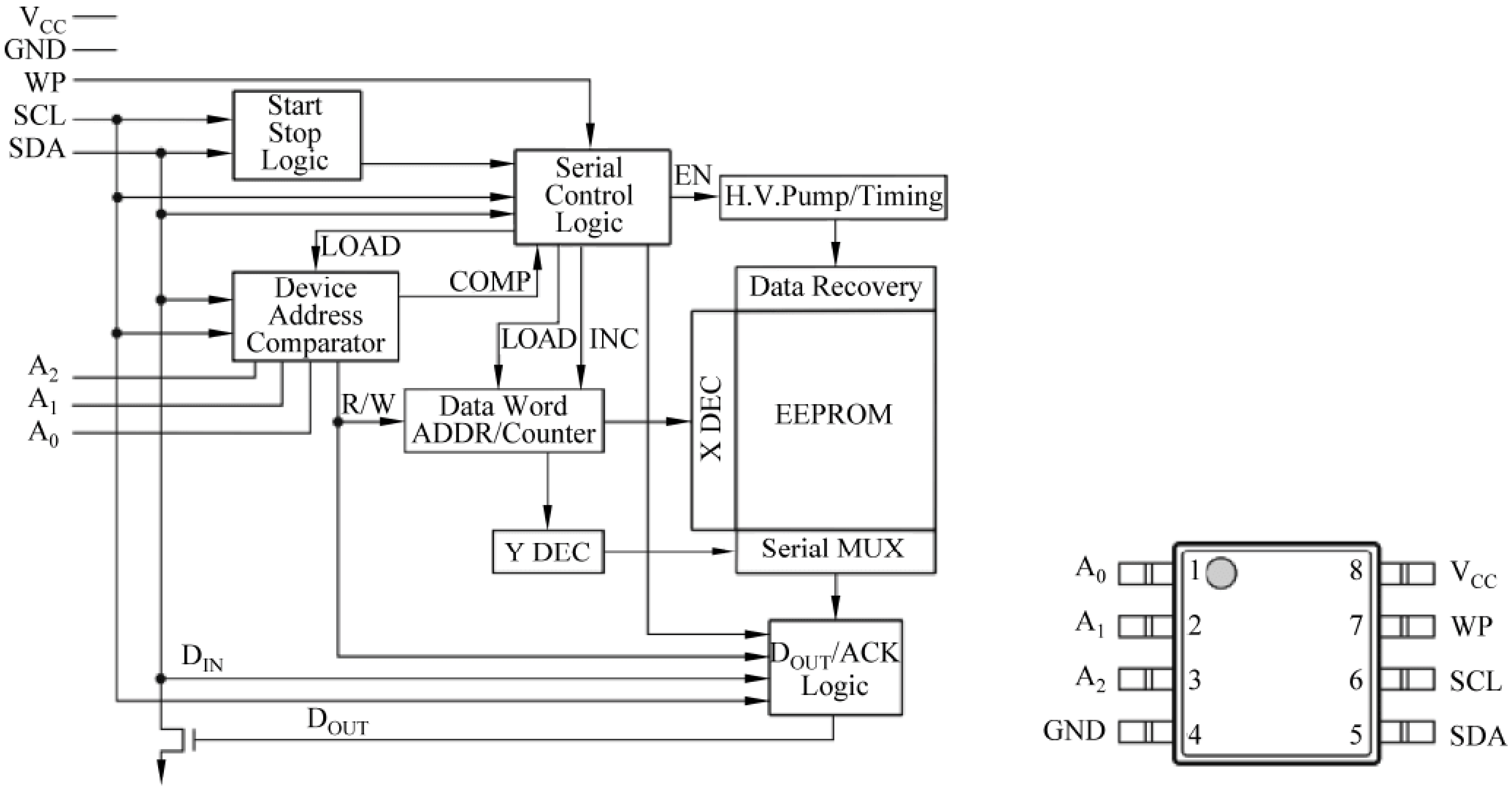


图 9-28 AT24C02 Flash 芯片

AT24C02 的操作时序见图 9-29。当需要往 Flash 写入数据时，我们首先发送 I2C 设备地址匹配 AT24C02，然后要发送 Flash 的写入地址，最后跟随一个或多个需要写入的数据，如图 9-29 的写单个存储字节和写多个存储字节所示。当需要从 Flash 读数据时，首先发送 I2C 地址匹配 AT24C02，此时我们需要先把要读的 Flash 地址写到 AT24C02，AT24C02 即可准备所要读的地址的数据，由于从设备无法主动发起数据，因此需要主设备再次发送 I2C 设备地址切换读写方向，发送一个读命令，从设备匹配地址后将准备好的数据发送到主设备，如图 9-29 所示的读单个字节和读多个字节时序。

程序实现的伪代码如下：

```
uint8_t E2promWriteByte( uint16_t flash_addr, uint8_t data )
{
    I2CStart();
```

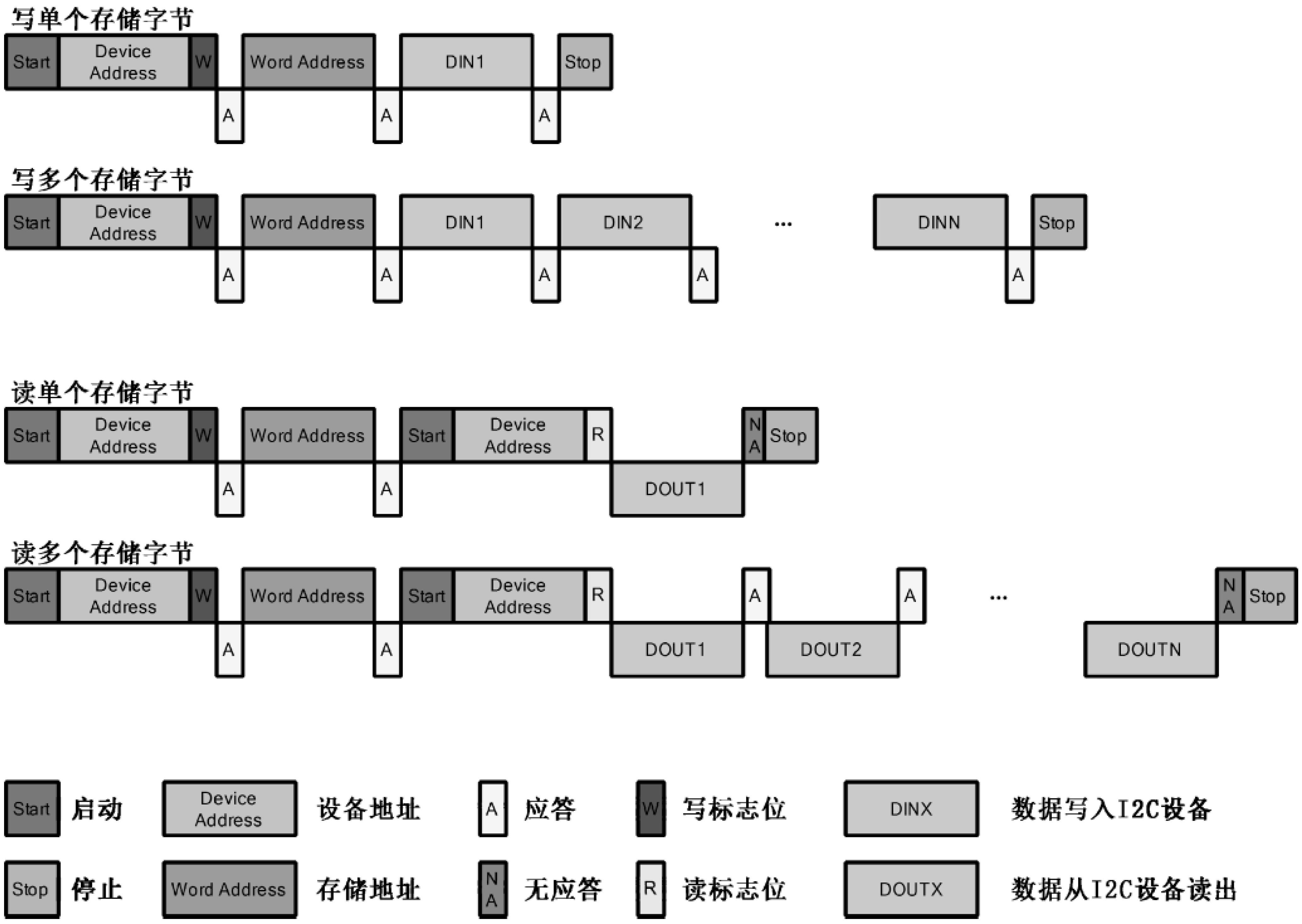


图 9-29　AT24C02 Flash 芯片读写时序

```
        I2CWriteByte( AT24C02ADDR_WRITE );
        WaitI2CSlaveAck();
        I2CWriteByte( flash_addr & 0xFF);
        WaitI2CSlaveAck();
        I2CWriteByte( data );
        WaitI2CSlaveAck();
        I2CStop();
        return 1;
}
uint8_t E2promReadByte( uint16_t flash_addr )
{
        unsigned char ReadValue;
        I2CStart();
        I2CWriteByte(AT24C02ADDR_WRITE );
        WaitI2CSlaveAck();
        I2CWriteByte( flash_addr & 0xFF );
        WaitI2CSlaveAck();
        I2CStart();                                          //切换方向
        I2CWriteByte(AT24C02ADDR_READ );
```

```
        WaitI2CSlaveAck();
        ReadValue = I2CReadByte();                        //读数据
        I2CStop();
        return ReadValue;
}
```

9.7.5 ADT7420 温度传感器通信

ADT7420 是一款 I2C 接口的数字温度传感器，可以通过 I2C 直接读取到转换后的温度值，其电路连接如图 9-30 所示，SCL 和 SDA 总线连接了两个上拉电阻，A0 和 A1 接到了低电平。ADT7420 设备地址的高四位为 1001，因此图 9-30 连接时，设备地址为 1001000。

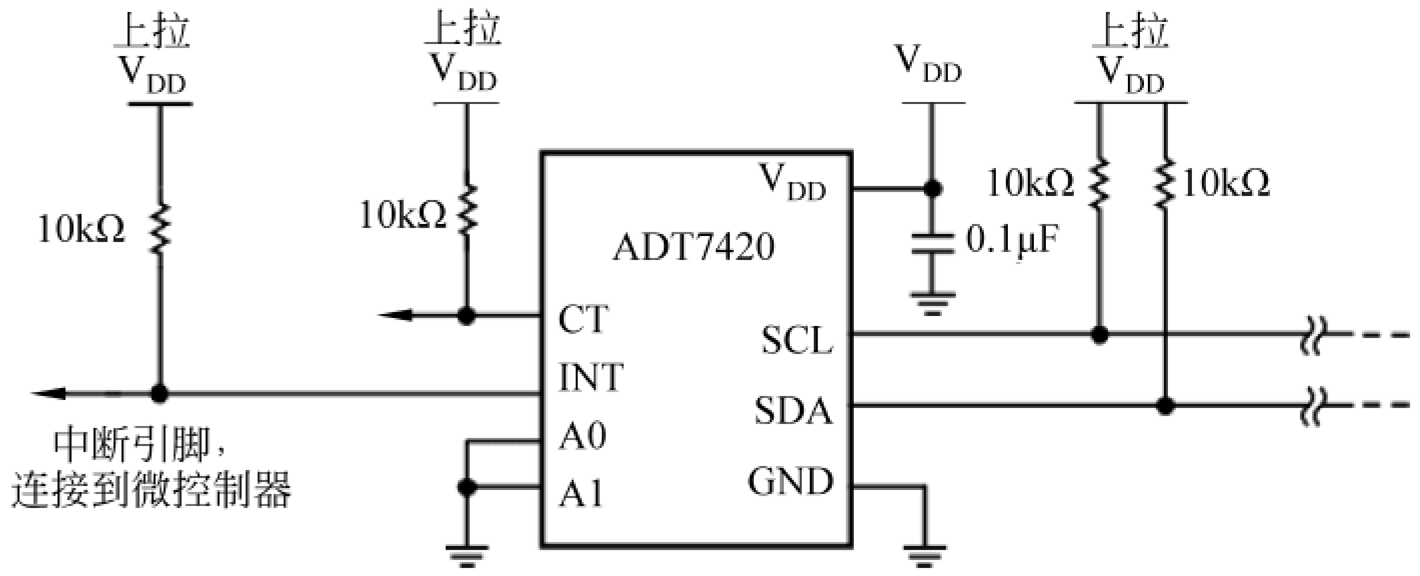


图 9-30　温度传感器 ADT7420 总线连接

ADT7420 内部有多个寄存器用于温度值的存储和传感器配置，如表 9-8 所示，要读写这些寄存器，需要指定寄存器的地址，这和 Flash 的读写时序类似。

表 9-8　ADT7420 寄存器

寄存器地址	描　　述	上电默认值
0x00	温度值最高有效字节	0x00
0x01	温度值最低有效字节	0x00
0x02	状态	0x00
0x03	配置	0x00
0x04	T_{HIGH} 设定点高有效字节	0x20(64℃)
0x05	T_{HIGH} 设定点低有效字节	0x00(64℃)
0x06	T_{LOW} 设定点高有效字节	0x05(10℃)
0x07	T_{LOW} 设定点低有效字节	0x00(10℃)
0x08	T_{CRIT} 设定点高有效字节	0x49(147℃)
0x09	T_{CRIT} 设定点低有效字节	0x80(147℃)

续表

寄存器地址	描　　述	上电默认值
0x0A	T_{HYST}设定点	0x05(5℃)
0x0B	ID	0xCB
0x2F	软件复位	0xXX

图 9-31 为对 ADT7420 的寄存器进行写的时序，首先发送 I2C 设备地址，方向为写，地址匹配后，主设备发送 8 位的寄存器地址，随后将需要写入该寄存器的数据发送给从设备，以 STOP 结束传输。

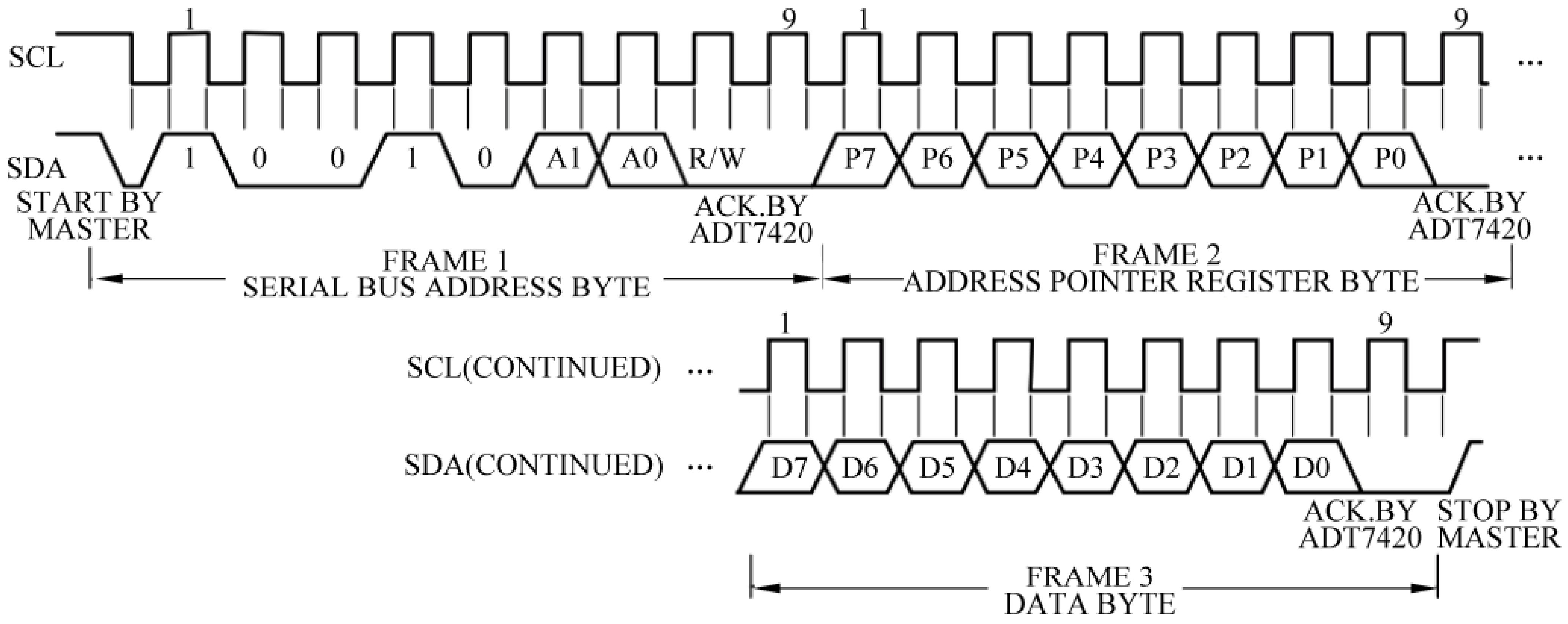


图 9-31　写寄存器时序

图 9-32 为对 ADT7420 的寄存器进行读的时序，首先发送 I2C 设备地址，方向为写，地址匹配后，主设备发送 8 位的寄存器地址，ADT7420 给出响应后，根据寄存器地址准备数据，主设备需要切换方向，重新发送 I2C 地址并将方向切换到读，ADT7420 匹配后将准备好

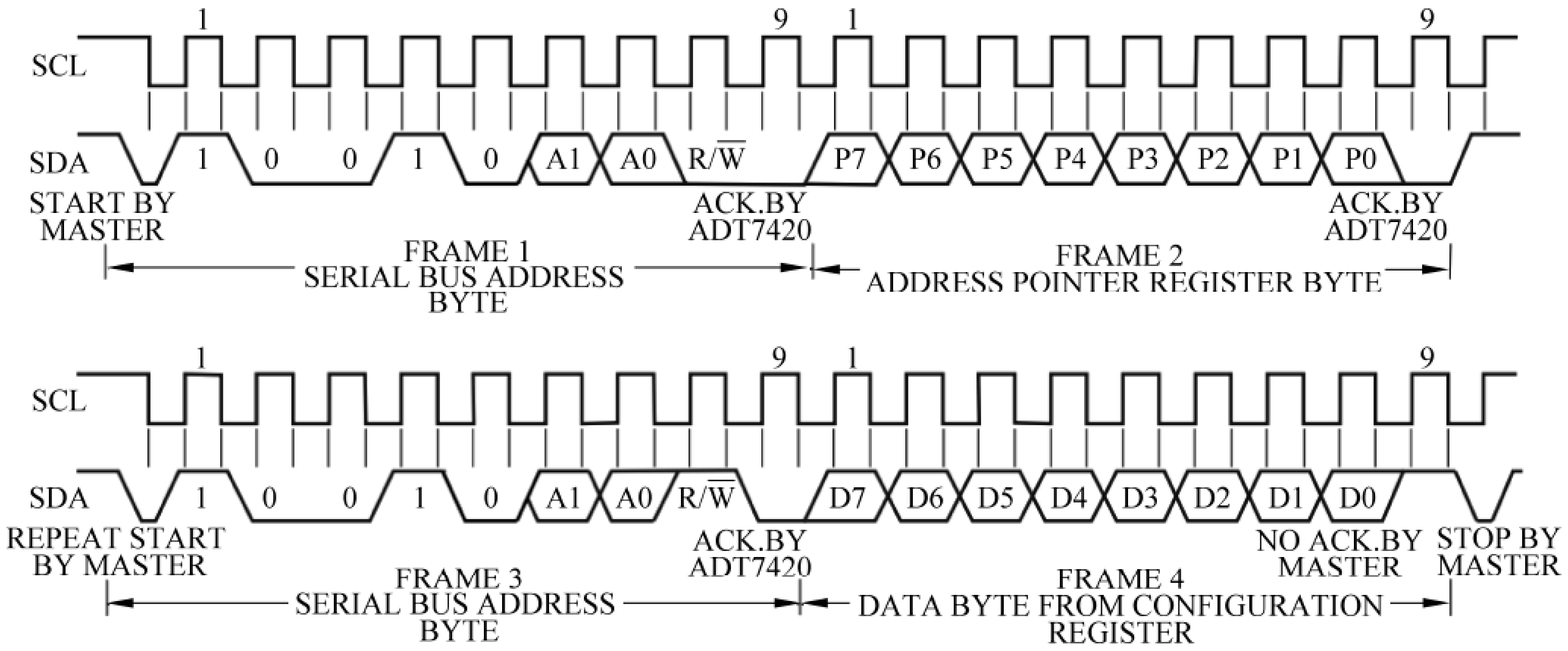


图 9-32　读寄存器时序

的数据发送给主设备,读完最后一个数据后,主设备发送一个 NACK 和 STOP,完成一次数据的读操作。

温度转换的函数如下:

```
float ADT7420_GetTemperature(void)
{
    uint8_t  msbTemp = 0;
    uint8_t  lsbTemp = 0;
    uint8_t  temp     = 0;
    float    tempC    = 0;
    msbTemp = ADT7420_GetRegisterValue(0x00);
    lsbTemp = ADT7420_GetRegisterValue(0x01);
    temp = ((uint16_t)msbTemp << 8) + lsbTemp;
    if(temp & 0x8000)                                    //负温度
        tempC = (float)((int32_t)temp - 65536) / 128;
    else
    tempC = (float)temp / 128;
    return tempC;
}
```

其中,读取温度寄存器的函数实现如下:

```
uint8_t ADT7420_GetRegisterValue(uint8_t registerAddress)
{
    uint8_t registerValue = 0;
    I2C_Write(ADT7420_ADDRESS, &registerAddress, 1, 0);
    //设备地址,寄存器地址,写入数据个数,是否发送停止
    I2C_Read(ADT7420_ADDRESS, &registerValue, 1, 1);
    //设备地址,读到的数据值,读数据的个数,是否发送停止位
    return registerValue;
}
```

I2C_Write 将 I2C 设备地址和寄存器地址写到 ADT7420,并不发送 STOP,I2C_Read 将 I2C 设备地址发送到 ADT7420,读取寄存器数据并发送 STOP 结束。

Chapter 10

第 10 章　SPI

【导读】 SPI 是快速同步串行总线，多用于数字外设之间的连接，本章首先介绍了 SPI 总线的概念和时序，然后对 STM32L152 的 SPI 总线控制器的内部结构，寄存器以及发送和接收流程进行了介绍，最后介绍了 CMSIS 提供的典型寄存器操作库函数。针对 SPI 总线时序传输，本章以温度传感器的操作时序为例进行了案例说明。

10.1　SPI 总线概述

SPI(Serial Peripheral Interface)总线是一种同步串行传输总线，允许主机以全双工与外围设备进行高速数据通信，主要用于嵌入式系统短距离通信。SPI 总线由摩托罗拉公司 80 年代后期提出，成为业界标准，但不同公司的处理器的实现细节可能有所不同，主要体现在寄存器定义、数据格式等。SPI 通常为四线制，因此也叫做四线串行总线，以区别单总线、双线和三线串行总线。

四线 SPI 支持全双工通信，采用由一个主设备管理的主从(master-slave)架构，主设备发起读写命令，通过片选信号与多个从设备进行通信。相对于 I2C 总线和 USART 总线，其主要优点为：

- 支持全双工通信。
- 总线驱动性能较好，可以支持 100MHz 以上的高速应用。
- 协议支持 8/16b 字长，可根据应用特点灵活选择字长。
- 硬件连接简单，只需四根信号线(也可支持三线传输)，相比 I2C 不需要仲裁。
- 从设备使用主设备时钟，且由片选选通从设备，无须寻址。

SPI 总线缺点如下：

- 片选选择不同从设备导致多从设备时需要更多的 I/O。
- 没有数据流控制，只能通过降低时钟匹配传输速度。
- 没有从设备接收数据 ACK。
- 典型应用只支持单主控。
- 相比于 RS-232、RS-422、RS-485 和 CAN，SPI 传输距离较短。

SPI 以其简单高效被大多数嵌入式处理器作为标准外设控制器集成到 MCU 中，一些典型外设也采用 SPI 总线接口，如 SD 卡、LCD 显示屏、串行 Flash 存储、RTC 芯片、振动、

压力等传感器。

10.2 SPI 总线控制器架构

10.2.1 接口信号和连接方式

SPI 协议定义四根信号线，分别为：

- SCK(Serial Clock)：串行时钟，作为主设备的输出，从设备的输入。
- MOSI(Master Output, Slave Input)：主设备输出/从设备输入，用于主模式发送数据，从模式接收数据。
- MISO(Master Input, Slave Output)：主设备输入/从设备输出，用于主模式接收数据，从模式发送数据。
- NSS(Slave Select)：从设备片选信号。

其中 MISO 方向为从设备到主设备，其余三个信号均为主设备到从设备。在支持三线 SPI 的控制器中，在三线双向模式下，主设备的 MOSI 和从设备的 MISO 作为双向 I/O 使用。

四线 SPI 主设备和从设备的硬件连接如图 10-1 所示。MOSI 脚相互连接，MISO 脚相互连接。这样，数据在主设备和从设备之间串行地传输。

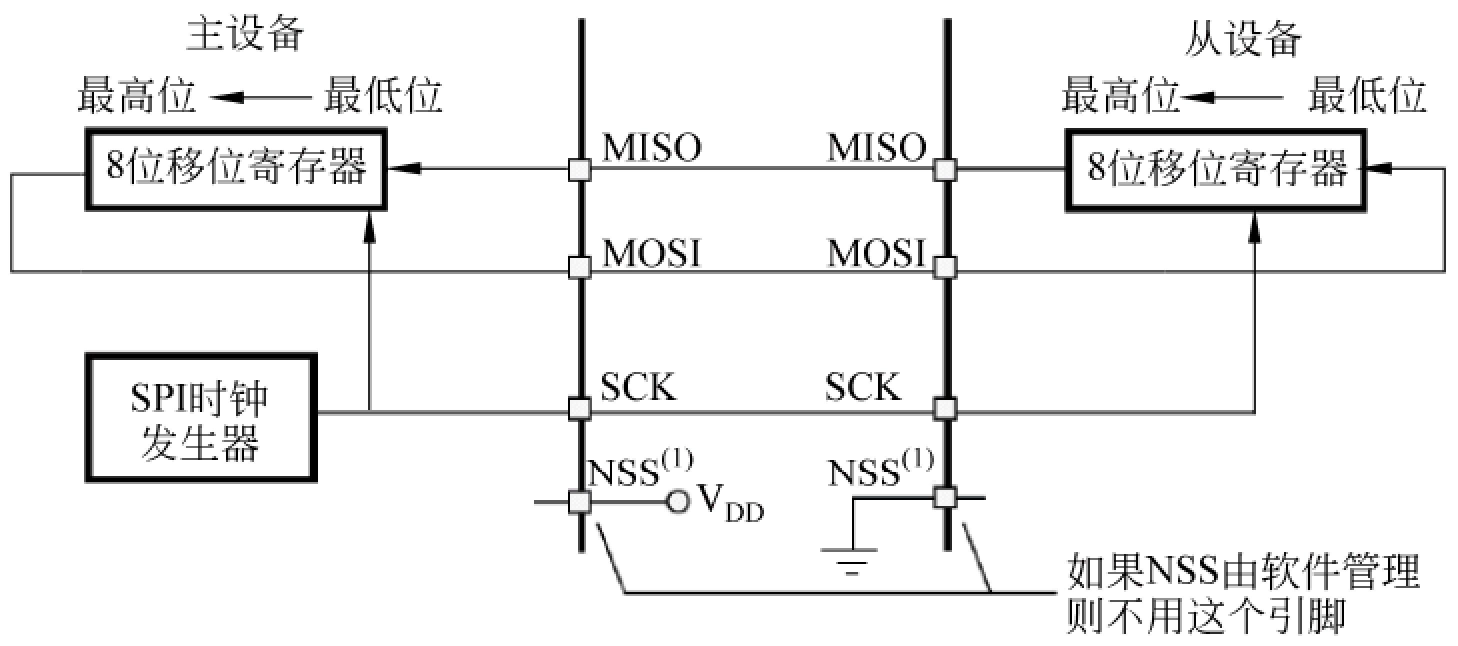


图 10-1 主从 SPI 设备的连接示意图

SPI 规定了两个 SPI 设备之间通信必须由主设备来控制从设备。从设备的时钟由主设备通过 SCK 引脚提供给从设备，从设备本身不能产生或控制 Clock。数据传输由主设备发起，主设备通过 MOSI 脚把数据发送给从设备，从设备通过 MISO 引脚回传数据，这意味全双工通信的数据输出和数据输入是用同一个时钟信号同步控制的。从设备在接收主设备的控制信号前，主设备首先通过 NSS 对从设备进行片选。图 10-1 中，只有两个设备连接，主设备的 NSS 连接到高电平，NSS 对于主设备不起作用，从设备的 NSS 连接到低电平，即一直选通该从设备。

一个主设备通过片选可以控制多个从设备，让主设备可以单独地与特定从设备通信，避免数据线上的冲突。当需要连接多个从设备时，可以通过多片选或者菊花链方式进行连接。

(1) 多片选连接方式：从设备的 NSS 引脚可以由主设备的任意一个标准 I/O 引脚来驱动，通常使用多个 I/O 口分别连接到从设备的 NSS 进行控制。如图 10-2 所示，所有从设备的 SCK、MOSI、MISO 都是连在一起，每个从设备都需要单独的片选信号，主设备每次只能选择其中一个从设备进行通信。由于每个设备都需要单独的片选信号，会占用较多的 I/O 资源，可以使用译码器电路或者采用菊花链方式。

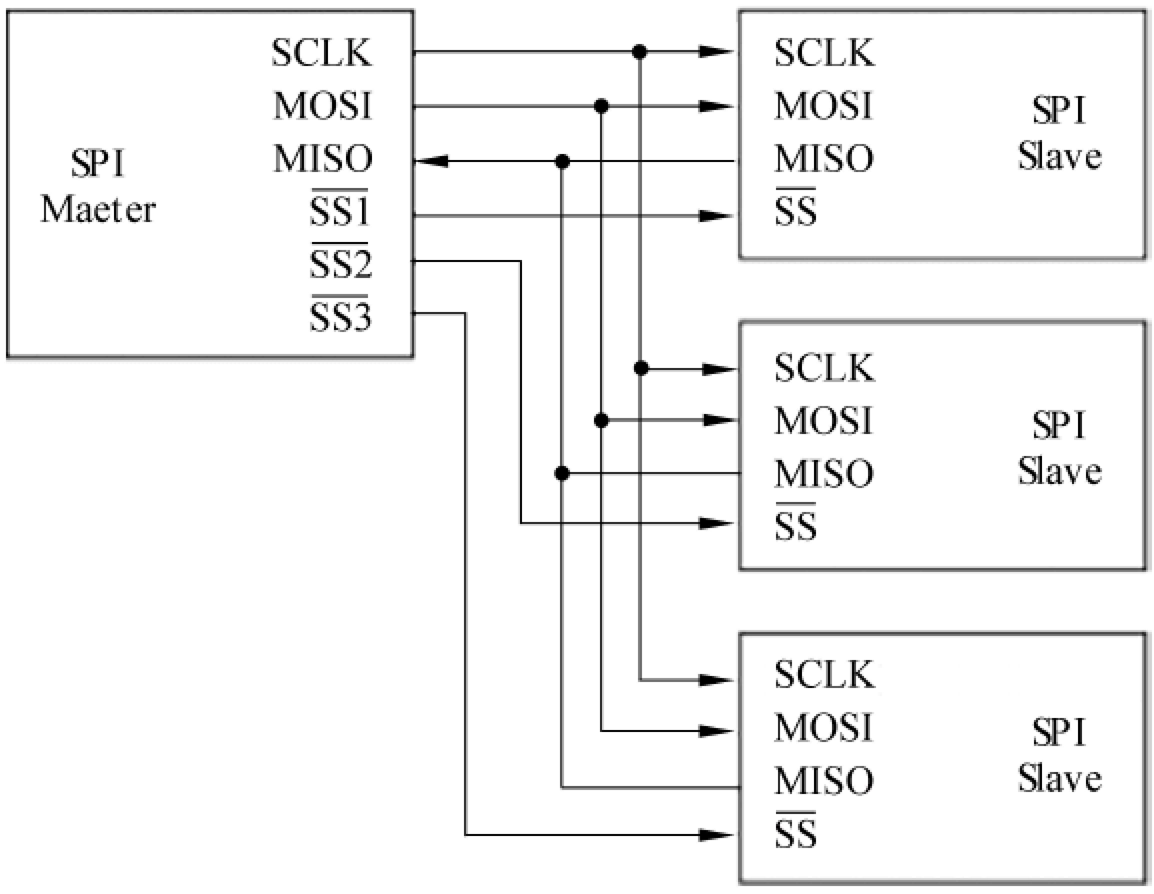


图 10-2　多片选方式控制多个从设备

(2) 菊花链连接方式：多片选方式占用较多的 I/O，只用一个 NSS 控制时，可以连接成如图 10-3 所示的电路，不同于图 10-2 的共享 MOSI 和 MISO 总线，菊花链方式下，所有从设备连接到一个 NSS 片选上，主设备 MOSI 连接到第一个从设备，第一个从设备的 MISO 连接到第二个从设备的 MOSI，依次连接，最后一个从设备的 MISO 连接到主设备的

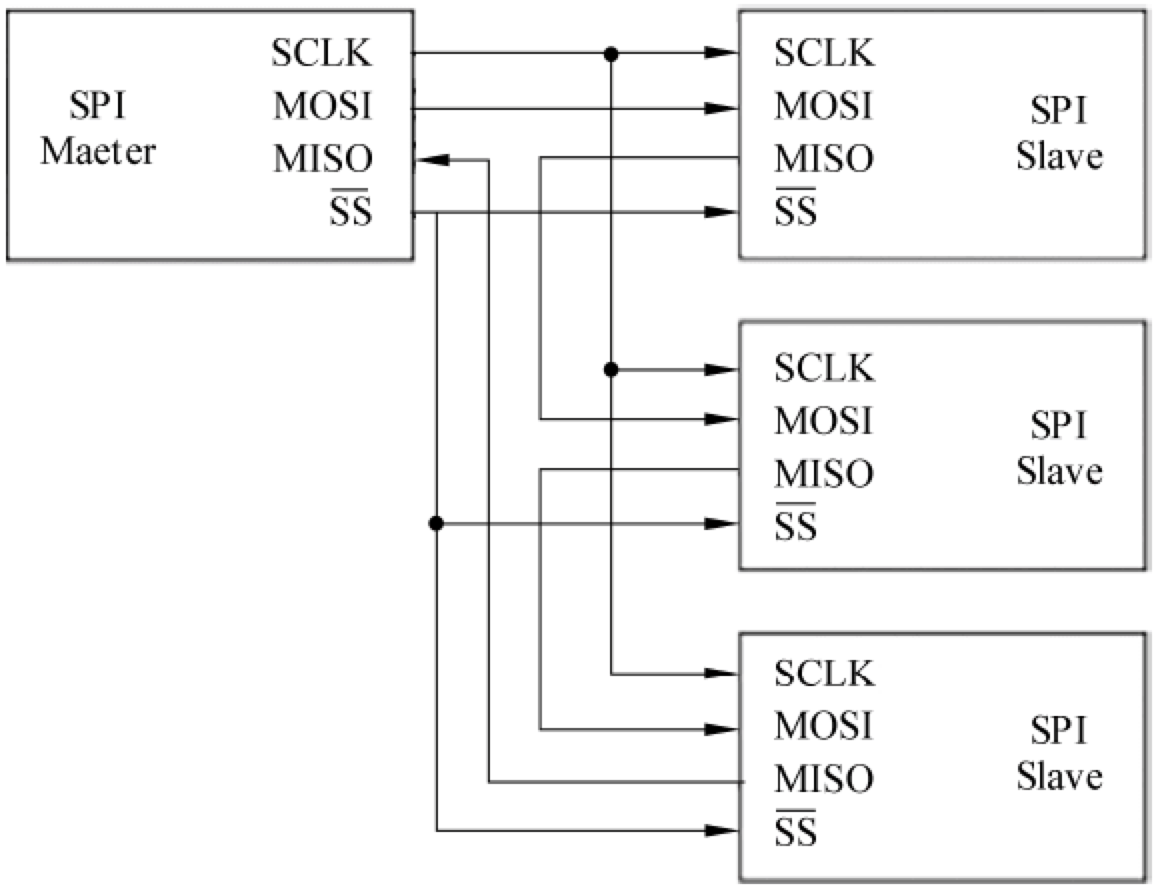


图 10-3　菊花链方式控制多个从设备

MISO,形成一个闭环。数据通过主设备发送,所有的从设备依次接收数据并向下传递。

由于 SPI 是全双工通信,因此,主从设备在数据通信过程中不能只充当一个发送者或者接收者,而是在每个时钟周期内主从设备都会发送并接收一个比特的数据,相当于设备间交换了一比特数据。其内部实现结构如图 10-4 所示,主从设备中均有一个移位寄存器用于存放所要传输的数据,在主设备的时钟控制下同步进行操作。在同一个时钟周期,主设备将自己移位寄存器中的最高位 MSB 通过 MOSI 送出,所有低位数据向高位移动 1 位,从设备将自己的最高位通过 MISO 送出,并将自己移位寄存器中的数据向高位移动 1 位,将 MOSI 上的来自主设备的数据存储到最低位;主设备采集 MISO 上的数据也存储到自己移位寄存器的最低位,这样循环 8 个时钟周期后,主从设备交换了一个字节。需要传输多个字节时,重复上述过程,传输完成后,通过片选信号释放从设备,这样主机的 MOSI 信号将被从设备忽略。单向传输时也保持上述流程,程序不处理从设备接收到的数据即可。

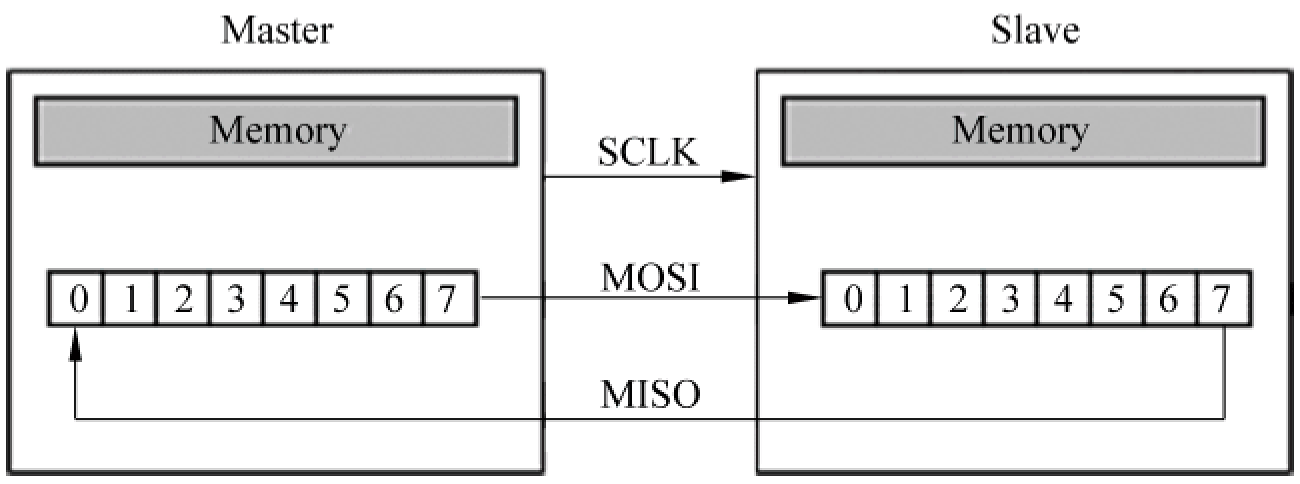


图 10-4　SPI 双向数据传输原理

表 10-1 为数据交换的时序案例。初始状态,主机发送数据 0xAA 到从机,从机发送数据 0x55 到主机,数据都送到了移位寄存器中。当第一个时钟边沿到达时,两个移位寄存器同时进行移位,将高位 MSB 的数据发送到数据线,在第二个时钟边沿到达时,将数据线 MOSI 和 MISO 同时存储到各自的移位寄存器,实现了一个数据位的交换。8 个时钟周期后,完成一个字节的数据交换。

表 10-1　数据交换时序

时钟脉冲	主机移位寄存器	从机移位寄存器	MISO	MOSI
0	10101010	01010101	0	0
1 上	0101010x	1010101x	0	1
1 下	01010100	10101011	0	1
2 上	1010100x	0101011x	1	0
2 下	10101001	01010110	1	0
3 上	0101001x	1010110x	0	1
3 下	01010010	10101101	0	1
4 上	1010010x	0101101x	1	0

续表

时钟脉冲	主机移位寄存器	从机移位寄存器	MISO	MOSI
4 下	10100101	01011010	1	0
5 上	0100101x	1011010x	0	1
5 下	01001010	10110101	0	1
6 上	1001010x	0110101x	1	0
6 下	10010101	01101010	1	0
7 上	0010101x	1101010x	0	1
7 下	00101010	11010101	0	1
8 上	0101010x	1010101x	1	0
8 下	01010101	10101010	1	0

10.2.2　传输模式和时序

在数据传输的过程中，主从设备必须在下一次数据传输之前将接收到的数据采样保存。SPI 的数据采样的时机和时钟的相位可以由两个控制信号 CPOL 和 CPHA 灵活进行配置。

CPOL(Clock Polarity)：决定在没有数据传输时时钟的空闲状态电平是高电平还是低电平，该信号为 1 时 SCK 引脚在空闲状态保持高电平，为 0 时 SCK 引脚在空闲状态保持低电平。

CPHA(Clock Phase)：定义 SPI 数据采样的时机，该信号为 1 时数据采样发生在时钟 SCK 的第二个边沿(CPOL 位为 0 时就是下降沿，CPOL 位为 1 时就是上升沿)，为 0 时数据采样发生在时钟 SCK 的第一个边沿(CPOL 位为 0 时就是上升沿，CPOL 位为 1 时就是下降沿)。

根据 CPOL 和 CPHA 的组合，将 SPI 可以分成 4 种传输模式，如表 10-2 所示，分别记为 SPI0、SPI1、SPI2 和 SPI3，其具体时序如图 10-5 所示。主从设备进行 SPI 通信时，要确保它们的传输模式设置相同。

表 10-2　四种传输模式

模　　式	CPOL	CPHA
SPI0	0	0
SPI1	0	1
SPI2	1	0
SPI3	1	1

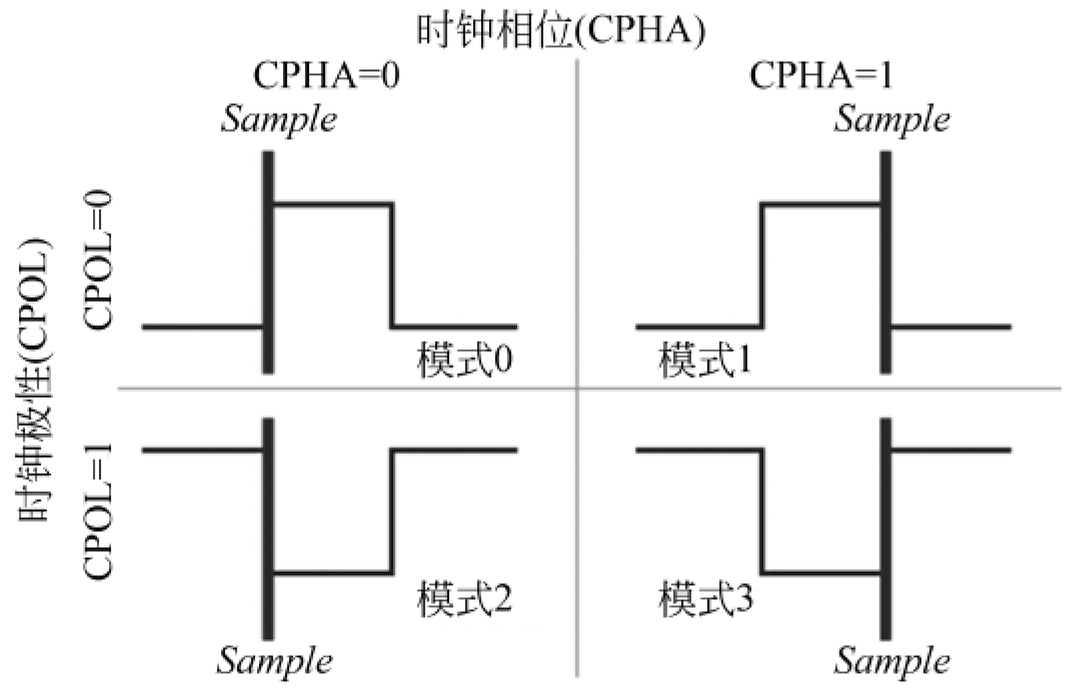


图 10-5　四种时序模式

图 10-6 为 SPI 传输的 4 种模式的时序。

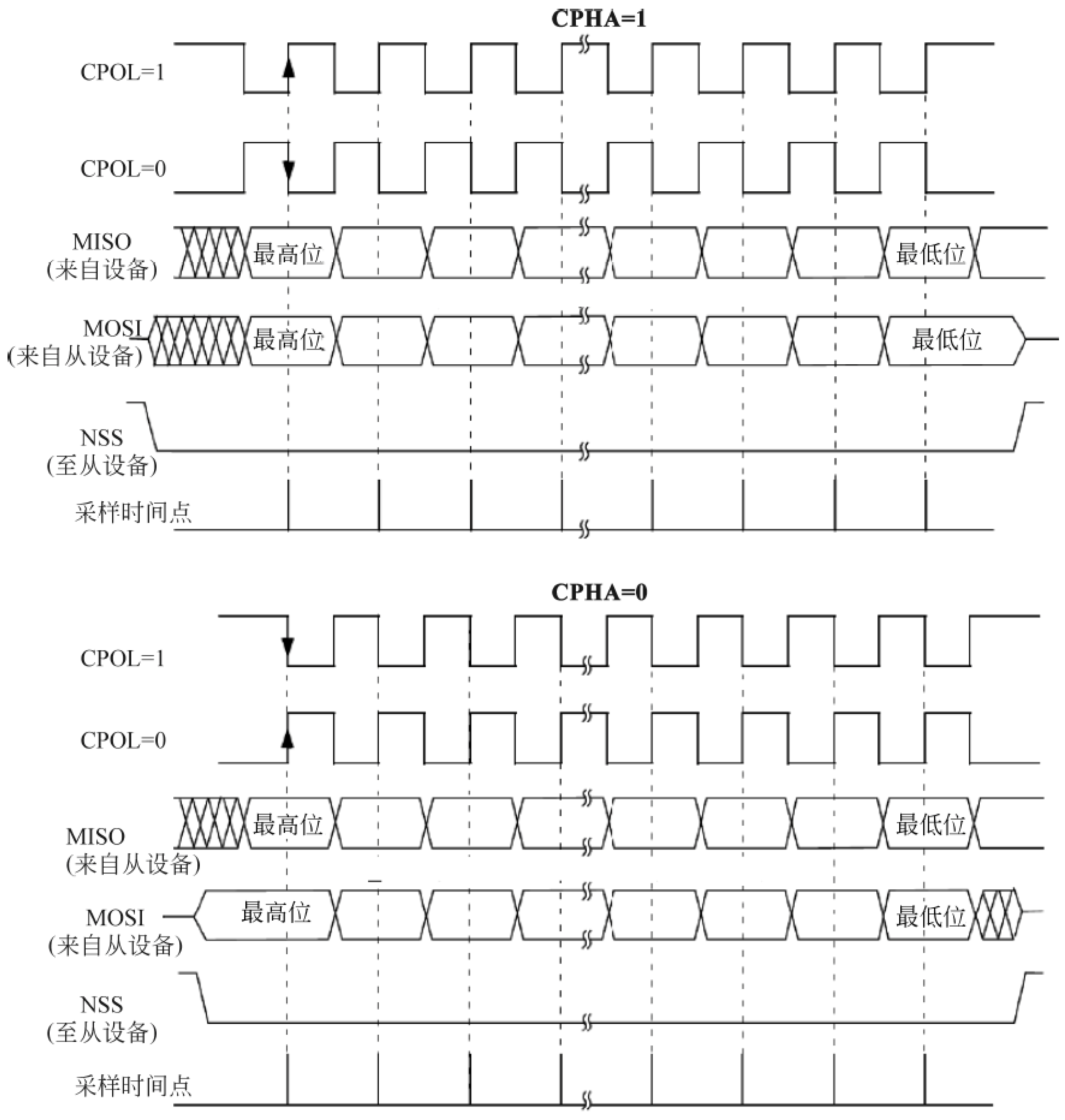


图 10-6　SPI 4 种模式下的采样时序

CPHA=1 时：

- 在 SCK 第二个时钟边沿采样和锁存数据。
- 在 SCK 第三个时钟边沿，将上个时钟边沿锁存的数据写入移位寄存器。
- 以此类推，数据在偶数边沿锁存，在奇数边沿写入移位寄存器。
- 经过 8/16 个时钟边沿后，串行传输的数据全部写入移位寄存器，完成主从设备的数据交换。

CPHA=0 时：

- SCK 的第一个时钟边沿采样和锁存数据。
- 在 SCK 的第二个时钟边沿，上个时钟边沿锁存的数据写入移位寄存器。
- 以此类推，数据在奇数边沿锁存，在偶数边沿写入移位寄存器。
- 经过 8/16 个时钟边沿后，串行传输的数据全部写入移位寄存器，完成主从设备的数据交换。

10.2.3　STM32L15x SPI 总线控制器

STM32L15x 系列微控制器有两个独立的 SPI 控制器，SPI1 连接在 APB2 总线上，SPI2 连接在 APB1 总线上，对于 STM32L15x 的 APB1 和 APB2 总线频率最高都支持到 32MHz，因此两个 SPI 控制器在最高通信速率上没有区别。

STM32L15x SPI 总线控制器支持 4 线和 3 线连接，8/16 位可选数据帧，支持多主模式，最大时钟频率为 APB 总线频率的 1/2，同时支持 8 种预分频系数。主从模式下 NSS 可以由软件或硬件管理，主从模式可动态切换，支持硬件 CRC，可编程数据传输顺序、DMA 传输以及专用的中断和总线状态标志位。其内部结构如图 10-7 所示。

如图 10-7 所示，SPI 控制器由接收和发送缓冲区，移位寄存器、波特率发生器、主控电路、通信电路以及控制和状态寄存器构成。波特率发生器用于控制时钟 SCK，其主要由 CR1 寄存器的分频因子 BR 和极性相、位控制信号 CPOL、CPHA 进行配置。主控电路用于确定 SPI 的双工/单工，3/4 线连接模式等，控制移位寄存器的时序，移位寄存器的数据输出次序（高位优先还是低位优先）由 CR1 寄存器的 LSBFIRST 决定。接收缓冲区用于存储从 MISO 发来的数据，发送缓冲区用于存储向 MOSI 总线发送的数据。通信电路用于管理片选信号 NSS、主从模式选择和 CRC 校验、中断和总线状态等。

STM32L15x 的每个 SPI 的 NSS 可以配置为输入，也可以配置为输出，可以通过 CR2 寄存器的 SSOE 控制。配置为输入时，NSS 的电平信号用于控制 SPI 控制器自己，配置为输出时，NSS 的信号发送给从设备进行片选，当配置为输出时，只有一个设备为主设备，其余设备均为从设备，不支持多主设备工作。SPI 控制器的外部接口 NSS 引脚连接到 SPI 控制器内部时的电路结构如图 10-8 所示，实际控制 SPI 片选的是内部 NSS，其有两个来源，分别为外部 NSS 引脚和内部寄存器 SSI 位，CR1 寄存器的 SSM 用于控制内部 NSS 的信号来源。

当 SSM 选通 SSI 作为内部 NSS 信号来源时，称为软件 NSS 管理模式，此时外部 NSS

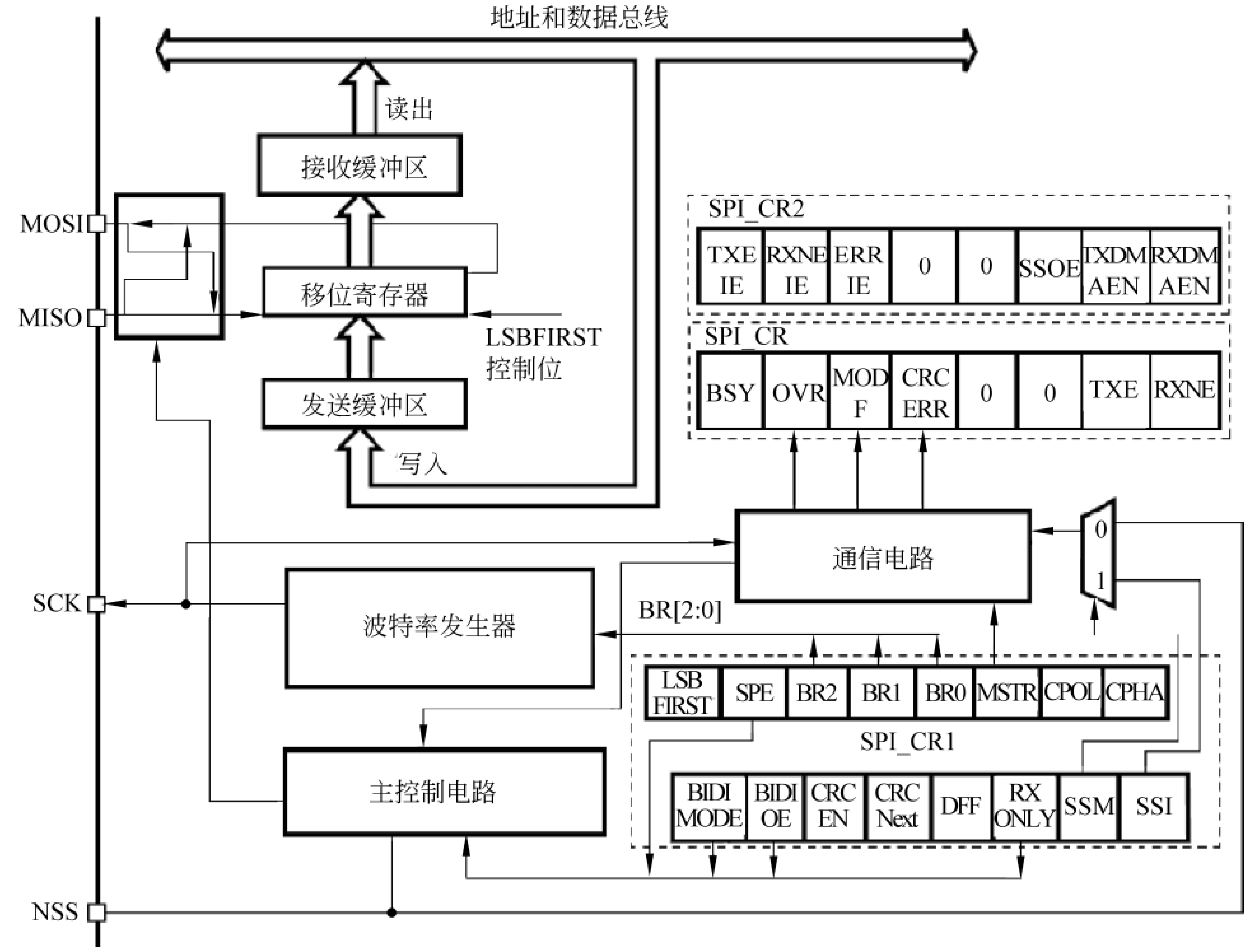


图 10-7　SPI 控制器内部结构图

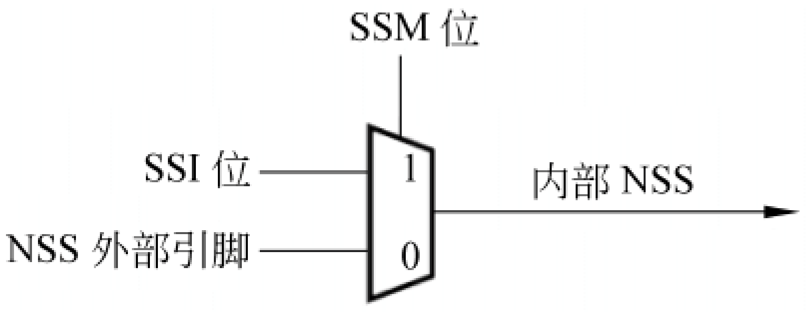


图 10-8　NSS 引脚选择内部电路

引脚无效，可以作为普通 GPIO 使用。当 SPI 工作在主模式时，SSI 需置为 1，当 SPI 工作在从设备模式下时，SSI 需配置为 0。例如，将微控制器集成的 SPI1 和 SPI2 控制器分别作为主设备和从设备连接进行数据传输，此时可配置 SPI1 SSI 位为 1，SPI2 的 SSI 为 0，无需连接 NSS 引脚。一般情况下，外设多为从设备，此时将 SPI 配置为主设备、软件管理模式，NSS 外部引脚作为 GPIO 控制从设备的片选，需要通过额外的 GPIO 控制对从设备进行片选。

当 SSM 选通 NSS 外部引脚作为内部 NSS 信号源时，称为 NSS 硬件管理模式。硬件管理模式下，当 NSS 配置为输出时，主设备一旦开始数据传输，自动将 NSS 置为 0，此时片选连接到 NSS 的其他设备均成为从设备接收数据（注意，NSS 信号不会自动变为 1）。硬件模式只用于多主设备下，当一个主设备发送数据时，它会自动拉低 NSS 信号，以通知所有其他

的设备它是主设备，如果它不能拉低 NSS，这意味着总线上有另外一个主设备在通信，这时将产生一个硬件失败错误(Hard Fault)。

10.3 SPI 寄存器说明

SPI 控制器的寄存器如表 10-3 所示。

表 10-3 SPI 寄存器列表

寄存器名称	偏移量	功 能	复位值
控制寄存器 1(SPI_CR1)	0x00	控制寄存器 1，用于配置帧格式、波特率、时钟以及三线、四线模式	0x0000 00C0
控制寄存器 2(SPI_CR2)	0x04	控制寄存器 2，中断和 DMA 管理	0x0000 0000
状态寄存器(SPI_SR)	0x08	状态寄存器，发送寄存器空、接收寄存器空等状态控制	0x0000 0000
数据寄存器(SPI_DR)	0x0C	发送和接收数据	0x0000 0000
CRC 多项式寄存器(SPI_CRC)	0x10	校验方法选择	0x0000 XXXX
RX CRC 寄存器(SPI_RXCRCR)	0x14	CRC 校验值	0x0000 0000
TX CRC 寄存器(SPI_TXCRCR)	0x18	CRC 校验值	0x0000 0000

1. SPI 控制寄存器 1(SPI_CR1)

控制寄存器 SPI_CR1 的有效域定义如图 10-9 所示。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BIDI MODE	BIDI OE	CRC EN	CRC NEXT	DFF	RX ONLY	SSM	SSI	*LSB FIRST*	SPE	BR [2:0]			MSTR	CPOL	CPHA
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 10-9 控制寄存器 CR1

BIDIMODE(Bidirectional Mode Enable)：单线双向数据模式使能域，用于配置 3 线/4 线 SPI 模式，0 表示双线单向模式，即全双工，1 表示单线双向模式，即半双工。

BIDIOE(Bidirectional Output Enable)：双向模式下的输出使能，和 BIDIMODE 位一起决定在单线双向模式下数据的输出方向，置为 0 时 SPI 为接收数据，置为 1 时发送数据。单线双向模式下，主设备 MOSI 连接到从设备 MISO 引脚，通过控制 BIDIOE 实现输入和输出的切换。

CRCEN(CRC Enable)：硬件 CRC 校验使能，置为 1 启用 CRC 计算。配置该位时需关闭 SPI(SPE 置为 0)，且该位只能在全双工模式下使用。

CRCNEXT(Transmit CRC next)：下一个发送 CRC，该位置为 1 时，SPI 将 CRC 寄存器中值通过移位寄存器发出，置为 0 时将发送缓冲区的数据发出。在启用 CRC 的情况下，当发送完最后一个数据后，将该位置 1，SPI 控制器自动发送 CRC 校验数据。

DFF(Data Frame Format)：数据帧格式域，该位置 0 表示使用 8 位数据帧格式进行发送/接收；置 1 表示使用 16 位数据帧格式进行发送/接收。配置该位时需要关闭 SPI。

RXONLY(Receive Only)：只接收，该位和 BIDIMODE 位一起决定在"双线单向"模式下的传输方向。置 0 表示全双工，置 1 表示只接收不发送。

SSM(Software Slave Management)：软件从设备管理，用于控制内部 NSS 信号的来源，置 1 时，启用软件从设备管理，内部 NSS 引脚上的电平由寄存器中 SSI 位的值决定。

SSI(Internal Salve Select)：内部从设备选择，该位只在 SSM 位为 1 时有效，用于配置内部 NSS 引脚的电平。

LSBFIRST：帧格式控制，0 表示先发送 MSB，1 表示先发送 LSB。

SPE(SPI Enable)：SPI 使能，置 0 禁止 SPI 设备，置 1 开启 SPI 设备。

BR[2：0](Buadrate Control)：波特率控制，三个二进制编码 000～111 分别表示 SPI 的 CLK 为 APB 总线时钟的 2、4、8、16、32、64、128、256 分频。

MSTR(Master Selection)：主设备选择，用于配置 SPI 工作在主设备模式(置为 1)还是从设备模式(置为 0)。

CPOL(Clock polarity)：时钟极性，和 CPHA 位组合用于选择 SPI 时序模式，该位为 0 表示空闲状态时 SCK 保持低电平，为 1 表示空闲状态时 SCK 保持高电平。

CPHA(Clock Phase)：时钟相位，该位置 0 表示数据采样从第一个时钟边沿开始；置 1 表示数据采样从第二个时钟边沿开始。

2. SPI 控制寄存器 2(SPI_CR2)

控制寄存器 SPI_CR2 的有效域定义如图 10-10 所示。

15–8	7	6	5	4	3	2	1	0
Reserved	TXEIE	RXNEIE	ERRIE	FRF	Res.	*SSOE*	TXDMAEN	RXDMAEN
	rw	rw	rw	rw		rw	rw	rw

图 10-10　控制寄存器 CR2

TXEIE(Tx Buffer Empty Interrupt Enable)：发送缓冲区空中断使能，用于配置 SPI 数据传输完成中断，置 0 表示禁止 TXE 中断，置 1 表示允许 TXE 中断。

RXNEIE(Rx Buffer Not Empty Interrupt Enable)：接收缓冲区非空中断使能，用于配置 SPI 数据接收中断，置 0 表示禁止 RXNE 中断，置 1 表示允许 RXNE 中断。

ERRIE(Error Interrupt Enable)：错误中断使能，用于控制当 SPI 传输出现校验错误、溢出、模式错误等时是否产生中断，0 禁止，1 允许。

SSOE(NSS Output Enable)：NSS 引脚的输入输出控制，0 表示 NSS 引脚为输入，1 表示 NSS 引脚为输出。

TXDMAEN(Tx DMA Enable)：发送缓冲区 DMA 使能，当该位被置 1 时，SPI_SR 寄存器中的发送完成标志 TXE 一旦被置 1 就发出 DMA 请求；该位置 0 不启用 DMA 传输。

RXDMAEN(Rx DMA Enable)：接收缓冲区 DMA 使能，当该位被置 1 时，SPI_SR 寄

存器中的 RXNE 标志一旦被置 1 就发出 DMA 请求;该位置 0 不启用 DMA 传输。

3. SPI 状态寄存器(SPI_SR)

控制寄存器 SPI_SR 的有效域定义如图 10-11 所示,包括:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved							FRE	BSY	OVR	MODF	CRC ERR	UDR	CHSIDE	TXE	RXNE
							r	r	r	r	rc_w0	r	r	r	r

图 10-11 状态寄存器 SR

BSY:忙标志,由硬件控制,该位为 0 表示 SPI 不忙,为 1 表示 SPI 正忙于通信,或者发送缓冲非空。SPI 开始传输时 BSY 被自动置 1,当传输结束、关闭 SPI 时被置 0,如果不是连续通信,在每个数据(8 位或 16 位)传输完成后之间,BSY 标志被自动置 0;当通信是连续时,主模式下 BSY 在整个传输期间保持为高电平,从模式下在每个数据传输完成之后被自动置 0,下一个数据开始时自动置 1。

OVR:溢出标志,当主设备已经发送了数据字节,而从设备还没有清除前一个数据字节产生的 RXNE 时,即为溢出错误。该位为 1 表示出现溢出错误,发生溢出错误时硬件自动置 1,需要软件进行清除。

MODF(Mode Fault):主模式失效错误,该位为 1 表示出现主模式被迫变更为从模式,发生模式错误时硬件自动置 1,需要软件进行清除。

CRCERR(CRC Error):CRC 错误标志,该位为 1 表示收到的 CRC 值和 SPI_RXCRCR 寄存器中的值不匹配。该位由硬件置位,由软件写 0 复位。

TXE(Tx Empty):发送缓冲为空,0 表示发送缓冲非空;1 表示发送缓冲为空,可以写下一个待发送的数据进入缓冲器中。当写入 SPI_DR 时,TXE 标志被清除。

RXNE(Rx Not Empty):接收缓冲非空,0 表示接收缓冲为空;1 表示在接收缓冲器中包含有效的接收数据。读 SPI 数据寄存器可以清除此标志。

FRE、UDR 和 CHSIDE 三个域在 SPI 模式下无效。

4. SPI 数据寄存器(SPI_DR)

数据寄存器如图 10-12 所示,DR 是一个 16 位寄存器,用于存储待发送或者已经收到的数据,数据寄存器 DR 实际对应两个缓冲区:一个用于写的发送缓冲和另外一个用于读的接收缓冲。写操作将数据写到发送缓冲区;读操作将返回接收缓冲区里的数据。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 10-12 数据寄存器 DR

寄存器 SPI_CR1 的 DFF 位对数据帧格式的选择,DFF 为 0 时数据帧格式为 8 位,发送和接收缓冲器只会用到 SPI_DR[7:0],其余位为 0。DFF 为 1 时,缓冲器使用整个数据寄存器 SPI_DR[15:0]。

5. SPI CRC 多项式寄存器(SPI_CRCPR)

CRC 多项式寄存器的有效域定义如图 10-13 所示,该寄存器包含了 CRC 计算时用到的多项式。其复位值为 0x0007,根据应用可以设置其他数值。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CRCPOLY[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 10-13　CRC 计算多项式寄存器

6. SPI Rx CRC 寄存器(SPI_RXCRCR)

接收数据 CRC 寄存器的有效域定义如图 10-14 所示,RXCRC[15:0]:接收 CRC 寄存器,在启用 CRC 计算时,RXCRC[15:0]中包含了依据收到的字节计算的 CRC 数值。当在 SPI_CR1 的 CRCEN 位写入'1'时,该寄存器被复位。CRC 计算使用 SPI_CRCPR 中的多项式。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RXCRC[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

图 10-14　接收数据 CRC 计算值

当数据帧格式被设置为 8 位时,仅低 8 位参与计算,并且按照 CRC8 的方法进行;当数据帧格式为 16 位时,寄存器中的所有 16 位都参与计算,并且按照 CRC16 的标准。

7. SPI Tx CRC 寄存器(SPI_TXCRCR)

发送数据 CRC 寄存器的有效域定义如图 10-15 所示,TxCRC[15:0]:发送 CRC 寄存器,在启用 CRC 计算时,TXCRC[15:0]中包含了依据将要发送的字节计算的 CRC 数值。当在 SPI_CR1 中的 CRCEN 位写入'1'时,该寄存器被复位。CRC 计算使用 SPI_CRCPR 中的多项式。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TXCRC[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

图 10-15　发送数据 CRC 计算值

10.4　SPI 通信流程

在进行 SPI 通信之前,首先要确定 SPI 的主从工作模式,一般情况下,MCU 被配置成主设备模式和其他外设进行数据通信,如果 MCU 作为从设备连接到其他设备上,需要将 MCU 的 SPI 配置为从设备模式。无论是 MCU 配置成主模式还是从模式,都需要根据所连接的从设备或主设备的配置进行参数配置。

10.4.1 SPI 双工通信模式配置

1. 从模式配置

在从设备模式下，SCK 引脚用于接收从主设备来的串行时钟，SPI_CR1 寄存器中 BR[2：0]的设置不影响数据传输速率。从设备模式的配置步骤如下：

- 设置 DFF 位以定义数据帧格式为 8 位或 16 位。
- 根据主设备的 CPOL 和 CPHA 时序要求，选择相同的 CPOL 和 CPHA 配置来定义数据传输和串行时钟之间的相位关系。
- 配置 SPI_CR1 寄存器中的 LSBFIRST 位确保帧格式与主设备相同。
- 配置 I/O 方向，MOSI 为输入，MISO 为输出，SPI_CR2 中的 SSOE 置为 0(NSS 引脚为输入)。
- 选择 NSS 的模式，若 NSS 连接到 GND 或主设备的 NSS 或 I/O 引脚，则选用硬件模式。若 NSS 未连接，则配置 NSS 软件模式，设置 SPI_CR1 寄存器中的 SSM 位并清除 SSI 位。
- 如果启用接收和发送中断，配置 SPI_CR2 寄存器的 TXEIE 和 RXNEIE，若启用 DMA 传输中断，配置 TXDMAIE 和 RXDMAIE。
- 清除 SPI_CR1 寄存器 MSTR 位、设置 SPE 位，启动 SPI 工作。

2. 主模式配置

在主设备模式下，主设备在 SCK 脚输出串行时钟。

- 通过 SPI_CR1 寄存器的 BR[2：0]位定义串行时钟波特率。
- 选择 CPOL 和 CPHA 位，定义数据传输和串行时钟间的相位关系。
- 设置 DFF 位来定义 8 位或 16 位数据帧格式。
- 配置 SPI_CR1 寄存器的 LSBFIRST 位定义帧格式。
- 配置引脚 I/O，MOSI 配置为输出，MISO 配置为输入。
- 配置 NSS 引脚的输入输出模式，输入硬件模式下，NSS 脚连接到高电平；输入软件模式下，需设置 SPI_CR1 寄存器的 SSM 位和 SSI 位；输出模式下，置 SSOE 位为 1。
- 设置 MSTR 位和 SPE 位，启动 SPI 传输。

3. 主从模式下的数据发送与接收流程

主模式下，当写入数据到 SPI_DR 寄存器(发送缓冲器)后，传输开始；在发送第一个数据位时，数据被并行地从发送缓冲器传送到 8 位的移位寄存器中，然后按顺序被串行地移位送到 MOSI 引脚上；MSB 在先还是 LSB 在先，取决于 SPI_CR1 寄存器中的 LSBFIRST 位的设置。数据从发送缓冲器传输到移位寄存器时 TXE 标志将被置位，如果设置了 SPI_CR1 寄存器中的 TXEIE 位，将产生中断。与此同时，在 MISO 引脚上接收到的数据，按顺序被串行地移位进入 8 位的移位寄存器中。

从模式下，当从设备接收到时钟信号并且第一个数据位出现在它的 MOSI 时，数据通信开始，MOSI 上传输第一个数据位时，发送缓冲器中的数据被并行地传送到移位寄存器，

随后 MOSI 的数据位依次移动进入移位寄存器，移位寄存器中的数据依次被发送到 MISO 引脚上；在 SPI 主设备开始数据传输之前，从设备需在发送寄存器中提前写入要发送的数据。当发送缓冲器中的数据传输到移位寄存器时，SPI_SP 寄存器的 TXE 标志被置 1，写入 SPI_DR 寄存器 TXE 被清 0。如果设置了 SPI_CR2 寄存器的 TXEIE 位，将会产生中断。

无论工作在何种模式，SPI 控制器在最后一个采样时钟边沿后，将移位寄存器中的数据传送到接收缓冲器，SPI_SR 寄存器中的 RXNE 标志被置 1，表明接收数据已就绪。读 SPI_DR 寄存器时，SPI 设备返回这个接收缓冲器的数值并将 RXNE 位置 0。如果设置了 SPI_CR2 寄存器中的 RXNEIE 位，则产生中断。

在发送和接收的过程中，SPI 处于通信状态是 BSY 标志被置 1，可以通过 BSY 信号判断传输是否完成。从模式下，一旦传输开始，如果下一个将发送的数据被放进了发送缓冲器，就可以维持一个连续的传输流，如图 10-16 所示，配置 LSB 优先，CPOL＝CPHA＝1，主设备发送 0xF1、0xF2、0xF3，从设备发送 0xA1、0xA2、0xA3。在将数据 0xF1 写入到数据寄存器后，TXE 标志被自动置 0，SPI 检测 SCK 时钟沿，在第一个下降沿，主模式下将会立即将 0xF1 写入发送缓冲器并将最低位发送到 MOSI 总线，BSY 标志被置 1，此时 SPI_DR 寄存器可以接收下一个将要被发送的数据，TXE 标志被自动置 1。在第一个上升沿，SPI 锁存 MISO 总线的 0xA1 的最低位电平信号，在第二个下降沿时，将 MISO 的信号写入移位寄存器最低位并在 MOSI 总线上送出 0xF1 的第二位，移位寄存器右移 1 位。在第 8 个时钟周期后，一个字节传输完成，BSY 标志被自动置 0，数据已被保存到接收缓冲区，RXNE 标志被自动置 1。第 9 个时钟周期，SPI 开始进行数据 0xF2 的发送和 0xA2 的接收。写发送缓冲

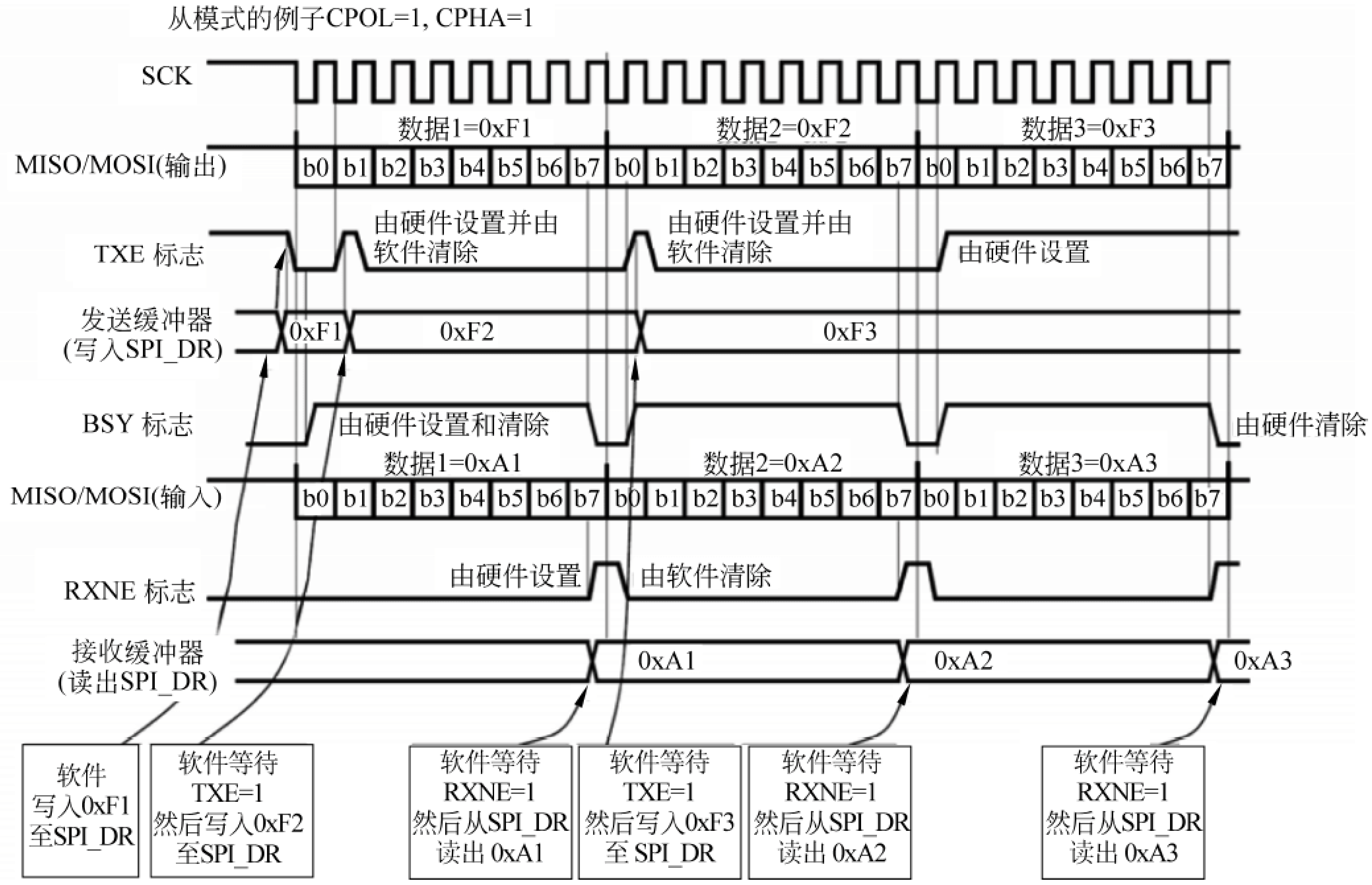


图 10-16　全双工从模式数据发送

器之前，需确认 TXE 标志应该为 1，否则新的数据会覆盖已经在发送缓冲器中的数据。主模式的时序和从模式类似，如图 10-17 所示，不同之处在于 BSY 信号在连续进行数据发送时一直保持高电平，直到所有数据发送结束，而从模式下 BSY 信号在每接收到一个主设备数据后被置为 0，下一个数据到达时又被置为 1。

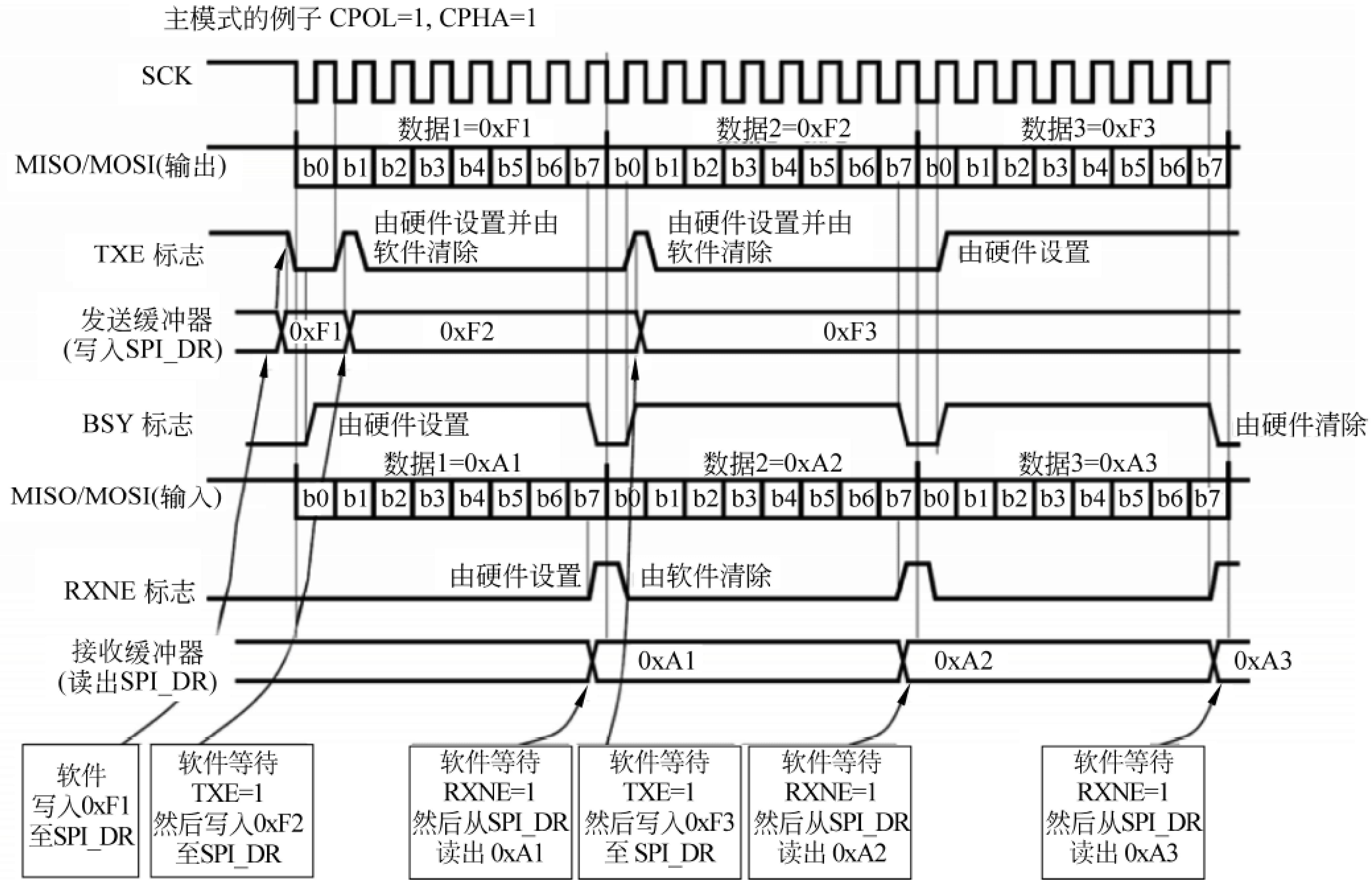


图 10-17 全双工主模式数据发送

全双工模式下发送和接收的处理流程如下：

- 设置 SPE 位为 1，使能 SPI 模块。
- 在 SPI_DR 寄存器中写入第一个要发送的数据。
- 等待 TXE＝1，然后写入第二个要发送的数据。
- 等待 RXNE＝1，然后读出 SPI_DR 寄存器并获得第一个接收到的数据，读 SPI_DR 的同时清除了 RXNE 位。重复步骤 3、4，发送后续的数据同时接收 $n-1$ 个数据。
- 等待 RXNE＝1，然后接收最后一个数据。
- 等待 TXE＝1，在 BSY＝0 之后关闭 SPI 模块。
- 如果开启了发送和接收中断，可以利用中断服务程序读写数据。

10.4.2 SPI 单工/半双工通信

在有些系统中，为节省 I/O 数量或者某些设备操作中主要以单向传输为主（如液晶屏），可以采用三线 SPI 接法。三线 SPI 只能实现单工/半双工通信，具体分为两种模式：

- 单线双向模式：1 条时钟线和 1 条双向数据线。
- 单线单向模式：1 条时钟线和 1 条单向数据线，只接收或只发送。

1. 单线双向模式

设置 SPI_CR1 寄存器中的 BIDIMODE 位为 1 启用单线双向模式。在这个模式下，SCK 引脚作为时钟，主设备使用 MOSI 引脚，从设备使用 MISO 引脚作为数据通信，即主设备 MOSI 和从设备 MISO 直接连接。传输的方向由 SPI_CR1 寄存器里的 BIDIOE 控制。当该位置 1 时数据线是输出（主设备发送数据），否则是输入（主设备接收数据）。BIDIMODE＝1 并且 BIDIOE＝1 时为双向发送，主设备数据线 MOSI 为输出，BIDIMODE＝1 并且 BIDIOE＝0 时为双向接收，主设备数据线 MOSI 为输入。

2. 单线单向模式

设置 SPI_CR1 寄存器中的 BIDIMODE 位为 0，且主设备 MOSI 连接到从设备 MISO 引脚时启用单线单向模式，在这个模式下，SPI 模块可以或者作为只发送，或者作为只接收，接收和发送状态由 RXONLY 位决定。

单向只发送模式（BIDIMODE＝0 并且 RXONLY＝0）：只发送模式和全双工模式类似，数据在发送引脚（主模式时是 MOSI、从模式时是 MISO）上传输，而接收引脚（主模式时是 MISO、从模式时是 MOSI）不使用，可以作为通用的 I/O 使用。程序中忽略接收缓冲器中的数据。

单向只接收模式（BIDIMODE＝0 并且 RXONLY＝1）：通过设置 SPI_CR2 寄存器的 RXONLY 位为 1 进入单向只接收模式，SPI 的输出功能被关闭，此时发送引脚（主模式时是 MOSI、从模式时是 MISO）可以作为 I/O 使用。在主设备中，一旦使能 SPI，即进入接收状态，BSY 标志始终置 1，关闭 SPI 时停止接收；在从设备中，一旦设备被片选（NSS＝0）且 SCK 有时钟脉冲，SPI 处于接收状态。

全双工模式是双线单向模式，和单工的单线单向模式的区别在于，在单线单向模式下，主设备 MOSI 和从设备 MISO 连接，双线单向模式下，主从设备的 MOSI、MISO 分别和 MOSI、MISI 连接。

3. 单线模式数据发送与接收流程

单线只发送过程使用 BSY 位等待传输的结束，如果数据连续发送，从模式下的时序流程如图 10-18 所示。

- 设置 SPE 位为 1，使能 SPI 模块。
- 在 SPI_DR 寄存器中写入第一个要发送的数据。
- 等待 TXE＝1，然后写入第二个要发送的数据；重复步骤 3，发送后续的数据。
- 写入最后一个数据到 SPI_DR 寄存器之后，等待 TXE＝1；然后等待 BSY＝0，完成数据传输。

主模式下的单线只发送和从模式下的区别在于在连续发送时 BSY 一直为高电平，不连续发送时与从模式单线只发送时序相同。

单向只接收模式的传输过程如图 10-19 所示，BSY 信号不起作用：

- 在 SPI_CR2 寄存器中，设置 RXONLY＝1。

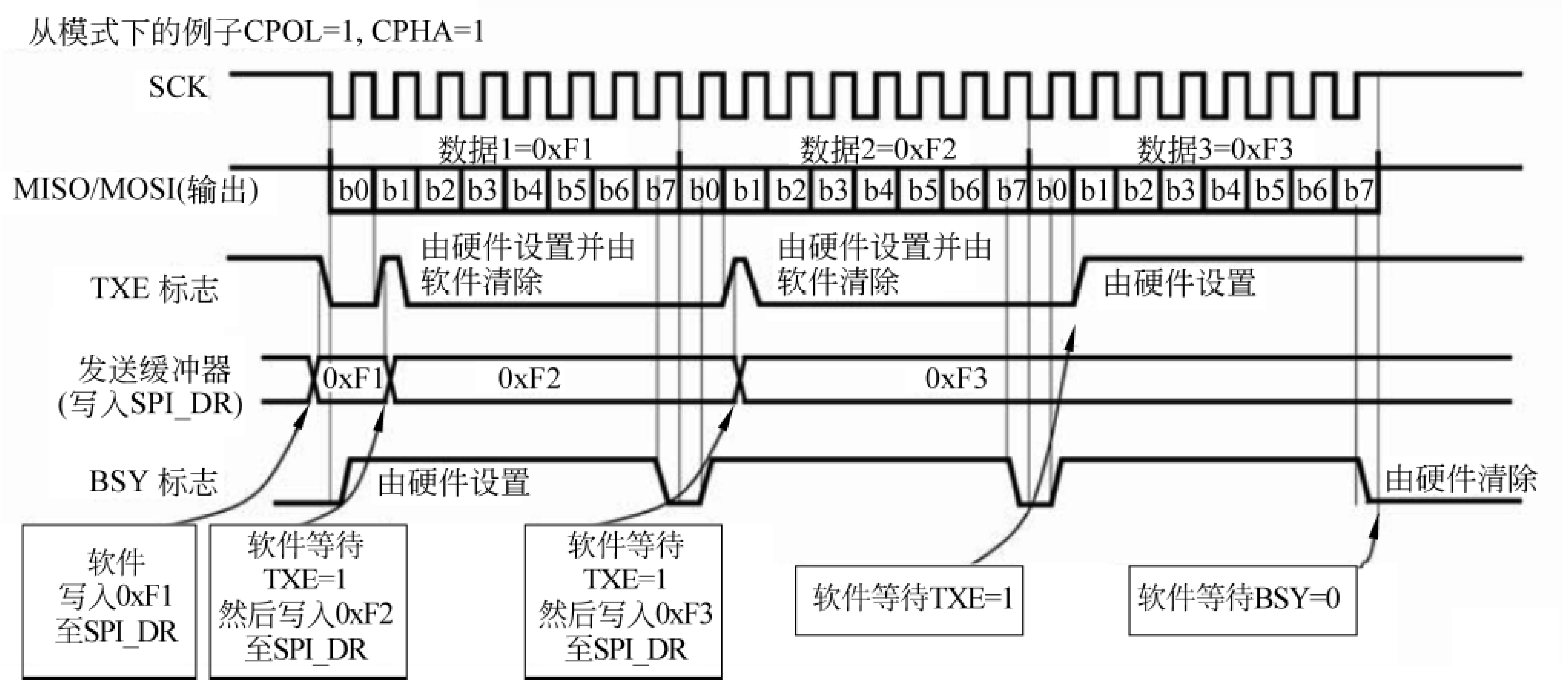


图 10-18　从模式单线只发送

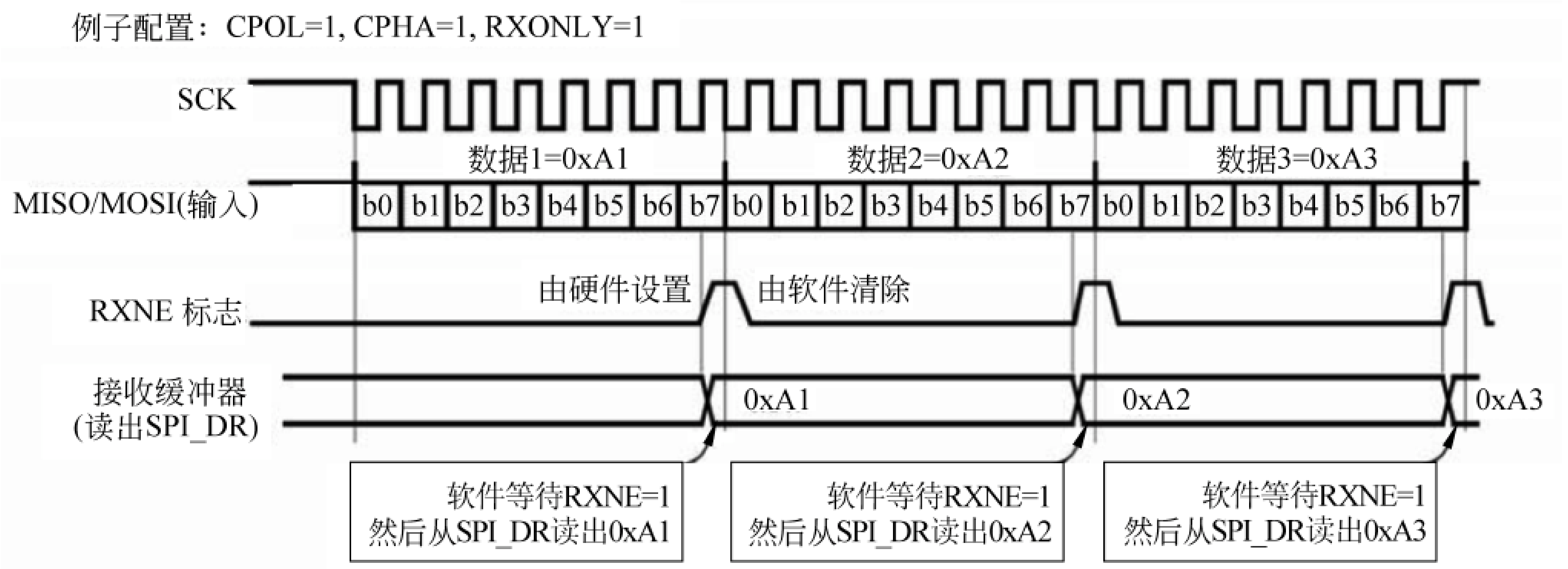


图 10-19　单线只接收模式

- 设置 SPE=1，使能 SPI 模块：
  - 主模式下立刻产生 SCK 信号，在关闭 SPI(SPE=0)之前不断地接收串行数据；
  - 从模式下，当 SPI 主设备拉低 NSS 信号并产生 SCK 时钟时，接收串行数据。
- 等待 RXNE=1，然后读出 SPI_DR 寄存器以获得收到的数据(同时会清除 RXNE 位)。重复此步骤接收所有数据。

单线双向传输的流程其发送过程和单线单向发送类似，接收过程和单线单向接收类似。

SPI 专用 GPIO 引脚如表 10-4 所示。

表 10-4　SPI GPIO 引脚

SPI 引脚功能	SPI1 I/O 引脚	SPI2 I/O 引脚	SPI3 I/O 引脚
SPI_NSS	PA4、PA15、PE12	PB12、PD0	PA15
SPI_SCK	PA5、PB3、PE13	PB13、PD1	PB3、PC10

续表

SPI 引脚功能	SPI1 I/O 引脚	SPI2 I/O 引脚	SPI3 I/O 引脚
SPI_MISO	PA6、PA11、PB4、PE14	PB14、PD3	PB4、PC11
SPI_MOSI	PA7、PA12、PB5、PE15	PB15、PD4	PB5、PC12

10.5 函数库

SPI 寄存器结构描述了固件函数库所使用的数据结构，固件库函数介绍了 ST 提供的典型库函数。

10.5.1 SPI 寄存器结构

SPI 寄存器结构，SPI_TypeDeff 在文件 stm32l1xx.h 中定义如下：

```
typedef struct
{
  __IO uint16_t CR1;                          // SPI 控制寄存器 1
  uint16_t RESERVED0;
  __IO uint16_t CR2;                          // SPI 控制寄存器 2
  uint16_t RESERVED1;
  __IO uint16_t SR;                           // SPI 状态寄存器
  uint16_t RESERVED2;
  __IO uint16_t DR;                           // SPI 数据寄存器
  uint16_t RESERVED3;
  __IO uint16_t CRCPR;                        // SPI CRC 多项式寄存器
  uint16_t RESERVED4;
  __IO uint16_t RXCRCR;                       // SPI 接收 CRC 寄存器
  uint16_t RESERVED5;
  __IO uint16_t TXCRCR;                       // SPI 发送 CRC 寄存器
  uint16_t RESERVED6;
} SPI_TypeDef;
```

2 个 SPI 外设声明文件 stm32l1xx.h：

```
#define PERIPH_BASE ((uint32_t)0x40000000)
#define APB1PERIPH_BASE PERIPH_BASE
#define APB2PERIPH_BASE (PERIPH_BASE + 0x10000)
#define AHBPERIPH_BASE (PERIPH_BASE + 0x20000)
#define SPI1_BASE (APB2PERIPH_BASE + 0x3000)
#define SPI2_BASE (APB1PERIPH_BASE + 0x3800)
```

```
#define SPI1 ((SPI_TypeDef * ) SPI1_BASE)
#define SPI2 ((SPI_TypeDef * ) SPI2_BASE)
```

SPI_InitTypeDef 定义于文件 stm32l1xx_spi.h,用于寄存器的初始化。

```
typedef struct
{
  uint16_t SPI_Direction;
  uint16_t SPI_Mode;
  uint16_t SPI_DataSize;
  uint16_t SPI_CPOL;
  uint16_t SPI_CPHA;
  uint16_t SPI_NSS;
  uint16_t SPI_BaudRatePrescaler;
  uint16_t SPI_FirstBit;
  uint16_t SPI_CRCPolynomial;
} SPI_InitTypeDef;
```

其中,SPI_Direction 设置了 SPI 单向或者双向的数据模式,取值为:

- SPI_Direction_2Lines_FullDuplex　　SPI 设置为双线双向全双工
- SPI_Direction_2Lines_RxOnly　　SPI 设置为双线单向接收
- SPI_Direction_1Line_Rx　　SPI 设置为单线双向接收
- SPI_Direction_1Line_Tx　　SPI 设置为单线双向发送

SPI_Mode 设置了 SPI 工作模式,取值为:

- SPI_Mode_Master　　设置为主 SPI
- SPI_Mode_Slave　　设置为从 SPI

SPI_DataSize 设置了 SPI 的数据大小,取值为:

- SPI_DataSize_16b　　SPI 发送接收 16 位帧结构
- SPI_DataSize_8b　　SPI 发送接收 8 位帧结构

SPI_CPOL 选择了串行时钟的默认值,取值为:

- SPI_CPOL_High　　时钟悬空高
- SPI_CPOL_Low　　时钟悬空低

SPI_CPHA 设置了位捕获的时钟活动沿的位置,取值为:

- SPI_CPHA_2Edge　　数据捕获于第二个时钟沿
- SPI_CPHA_1Edge　　数据捕获于第一个时钟沿

SPI_NSS 指定了 NSS 信号由硬件(NSS 引脚)还是软件(使用 SSI 位)管理,取值为:

- SPI_NSS_Hard　　NSS 由外部引脚管理
- SPI_NSS_Soft　　内部 NSS 信号有 SSI 位控制

SPI_BaudRatePrescaler 用来定义波特率预分频的值,这个值用以设置发送和接收的 SCK 时钟,取值为 SPI_BaudRatePrescalerX,其中 X 取值范围为 2、4、8、16、32、64、128、256,分别表示波特率预分频值为 X。

SPI_FirstBit 指定了数据传输从 MSB 位还是 LSB 位开始，取值为：

- SPI_FisrtBit_MSB　　数据传输从 MSB 位开始
- SPI_FisrtBit_LSB　　数据传输从 LSB 位开始

SPI_CRCPolynomial 定义了用于 CRC 值计算的多项式，默认值为 7。

10.5.2　SPI 库函数

SPI 库函数列表如表 10-5 所示。

表 10-5　SPI 库函数列表

函　数　名	描　　述
SPI_I2S_DeInit	将外设 SPIx 寄存器重设为默认值
SPI_Init	根据 SPI_InitStruc 中指定的参数初始化外设 SPIx 寄存器
SPI_StructInit	把 SPI_InitStruct 中的参数按默认值填入
SPI_DataSizeConfig	配置 8 位/16 位数据传输
SPI_Cmd	使能或者失能 SPI 外设
SPI_I2S_ITConfig	使能或者失能指定的 SPI 中断
SPI_DMACmd	使能或者失能指定 SPI 的 DMA 请求
SPI_I2S_SendData	通过外设 SPIx 发送一个数据
SPI_I2S_ReceiveData	返回通过 SPIx 最近接收的数据
SPI_NSSInternalSoftwareConfig	为选定的 SPI 软件配置内部 NSS 引脚
SPI_SSOutputCmd	使能或者失能指定的 SPI SS 输出
SPI_BiDirectionalLineConfig	选择指定 SPI 在双向模式下的数据传输方向
SPI_I2S_GetFlagStatus	检查指定的 SPI 标志位设置与否
SPI_I2S_ClearFlag	清除 SPIx 的待处理标志位
SPI_I2S_GetITStatus	检查指定的 SPI 中断发生与否
SPI_I2S_ClearITPendingBit	清除 SPIx 的中断待处理位
SPI_CalculateCRC	计算 CRC
SPI_TransmitCRC	传输 CRC
SPI_GetCRC	获取接收 CRC

在操作 SPI 控制器之前，需要打开 SPI 总线时钟，对 SPI1，调用 RCC_APB2PeriphClock-Cmd()，对 SPI2 调用 RCC_APB1PeriphClockCmd()。

1）函数 SPI_I2S_DeInit

功能描述：将外设 SPIx 寄存器重设为默认值。

函数原型：void SPI_I2S_DeInit(SPI_TypeDef * SPIx)。

输入参数 SPIx 用来选择 SPI 外设。

示例：

```
SPI_I2S_DeInit(SPI2);
```

2）函数 SPI_Init

功能描述：根据 SPI_InitStruct 中指定的参数初始化外设 SPIx 寄存器。

函数原型：void SPI_Init(SPI_TypeDef * SPIx, SPI_InitTypeDef * SPI_InitStruct)。

输入参数 SPIx 用来选择 SPI 外设，SPI_InitStruct 为指向结构 SPI_InitTypeDef 的指针，包含了外设 SPI 的配置信息。

示例：

```
SPI_InitTypeDef SPI_InitStructure;
SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
SPI_InitStructure.SPI_DatSize = SPI_DatSize_16b;
SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low;
SPI_InitStructure.SPI_CPHA = SPI_CPHA_2Edge;
SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_128;
SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
SPI_InitStructure.SPI_CRCPolynomial = 7;
SPI_Init(SPI1, &SPI_InitStructure);
```

3）函数 SPI_StructInit

功能描述：把 SPI_InitStruct 中的每一个参数按默认值填入。

函数原型：void SPI_StructInit(SPI_InitTypeDef * SPI_InitStruct)。

输入参数：SPI_InitStruct：指向结构 SPI_InitTypeDef 的指针，待初始化。默认值为：

```
SPI_Direction                SPI_Direction_2Lines_FullDuplex
SPI_Mode                     SPI_Mode_Slave
SPI_DataSize                 SPI_DataSize_8b
SPI_CPOL                     SPI_CPOL_Low
SPI_CPHA                     SPI_CPHA_1Edge
SPI_NSS                      SPI_NSS_Hard
SPI_BaudRatePrescaler        SPI_BaudRatePrescaler_2
SPI_FirstBit                 SPI_FirstBit_MSB
SPI_CRCPolynomial
```

示例：

```
SPI_InitTypeDef SPI_InitStructure;
SPI_StructInit(&SPI_InitStructure);
```

4）函数 SPI_Cmd

功能描述：使能或者失能 SPI 外设。

函数原型：void SPI_Cmd(SPI_TypeDef * SPIx, FunctionalState NewState)。

输入参数 SPIx 用于选择 SPI 外设，NewState 用于设置外设 SPIx 的状态，取值为 ENABLE 或者 DISABLE。

示例：

```
SPI_Cmd(SPI1, ENABLE);
```

5）函数 SPI_I2S_ITConfig

功能描述：使能或者失能指定的 SPI 中断。

函数原型：void SPI_I2S_ITConfig(SPI_TypeDef * SPIx, uint16_t SPI_I2S_IT, FunctionalState NewState)。

输入参数 SPIx 用来选择 SPI 外设，SPI_I2S_IT 为待使能或者失能的 SPI 中断源，其取值为：

- SPI_IT_TXE　　发送缓存空中断屏蔽
- SPI_IT_RXNE　　接收缓存非空中断屏蔽
- SPI_IT_ERR　　错误中断屏蔽

输入参数 NewState 为 SPIx 中断的状态，取值为 ENABLE 或者 DISABLE。

示例：

```
SPI_I2S_ITConfig(SPI2, SPI_IT_TXE, ENABLE);
```

6）函数 SPI_I2S_SendData

功能描述：通过外设 SPIx 发送一个数据。

函数原型：void SPI_I2S_SendData(SPI_TypeDef * SPIx, uint16_t Data)。

输入参数 SPIx 用来选择 SPI 外设，Data 为待发送的数据。

示例：

```
SPI_I2S_SendData(SPI1, 0xA5);
```

7）函数 SPI_I2S_ReceiveData

功能描述：返回通过 SPIx 最近接收的数据。

函数原型：uint16_t SPI_I2S_ReceiveData(SPI_TypeDef * SPIx)。

输入参数 SPIx 用来选择 SPI 外设。

示例：

```
uint16_t ReceivedData;
ReceivedData = SPI_I2S_ReceiveData(SPI2);
```

8）函数 SPI_NSSInternalSoftwareConfig

功能描述：为选定的 SPI 进行软件内部 NSS 引脚配置。

函数原型：void SPI_NSSInternalSoftwareConfig(SPI_TypeDef * SPIx, uint16_t SPI_NSSInternalSoft)。

输入参数 SPIx 用于选择 SPI 外设,SPI_NSSInternalSoft 为 SPI NSS 内部状态,其取值范围为：

- SPI_NSSInternalSoft_Set　　内部设置 NSS 引脚高电平
- SPI_NSSInternalSoft_Reset　　内部重置 NSS 引脚低电平

示例：

```
SPI_NSSInternalSoftwareConfig(SPI1, SPI_NSSInternalSoft_Set);
SPI_NSSInternalSoftwareConfig(SPI2, SPI_NSSInternalSoft_Reset);
```

9）函数 SPI_SSOutputCmd

功能描述：使能或者失能指定的 SPI NSS 引脚输出。

函数原型：void SPI_SSOutputCmd(SPI_TypeDef * SPIx, FunctionalState NewState)。

输入参数 SPIx 用来选择 SPI 外设,NewState 为 SPI NSS 引脚输出的状态,取值为 ENABLE 或者 DISABLE。

示例：

```
SPI_SSOutputCmd(SPI1, ENABLE);
```

10）函数 SPI_BiDirectionalLineConfig

功能描述：选择指定 SPI 在双向模式下的数据传输方向。

函数原型：SPI_BiDirectionalLineConfig(SPI_TypeDef * SPIx, uint16_t SPI_Direction)。

输入参数：SPIx 用来选择 SPI 外设,SPI_Direction 选择 SPI 在双向模式下的数据传输方向,取值为：

- SPI_Direction_Tx　　选择 Tx 发送方向
- SPI_Direction_Rx　　选择 Rx 接受方向

示例：

```
SPI_BiDirectionalLineConfig(SPI_Direction_Tx);
```

11）函数 SPI_I2S_GetFlagStatus

功能描述：检查指定的 SPI 标志位设置与否。

函数原型：FlagStatus SPI_GetFlagStatus(SPI_TypeDef * SPIx, uint16_t SPI_I2S_FLAG)。

输入参数 SPIx 用于选择 SPI 外设,SPI_I2S_FLAG 为待检查的 SPI 标志位,包括：

- SPI_FLAG_BSY　　忙标志位
- SPI_FLAG_OVR　　超出标志位

- SPI_FLAG_MODF　　模式错位标志位
- SPI_FLAG_CRCERR　　CRC 错误标志位
- SPI_FLAG_TXE　　发送缓存空标志位
- SPI_FLAG_RXNE　　接受缓存非空标志位

返回值为 SPI_I2S_FLAG 的状态，值为 SET 或者 RESET。

12）函数 SPI_I2S_ClearFlag

功能描述：清除 SPIx 的待处理标志位。

函数原型：void SPI_I2S_ClearFlag(SPI_TypeDef * SPIx，uint16_t SPI_I2S_FLAG)。

输入参数 SPIx 用于选择 SPI 外设，SPI_I2S_FLAG 为待清除的 SPI 标志位，取值与 SPI_I2S_GetFlagStatus 的 SPI_I2S_FLAG 相同，但是，不能清除标志位 BSY，TXE 和 RXNE，这三个标志位由硬件清零。

示例：

```
SPI_I2S_ClearFlag(SPI2, SPI_FLAG_OVR);
```

13）函数 SPI_I2S_GetITStatus

功能描述：检查指定的 SPI 中断发生与否。

函数原型：ITStatus SPI_I2S_GetITStatus(SPI_TypeDef * SPIx，uint8_t SPI_I2S_IT)。

输入参数 SPIx 用来选择 SPI 外设，SPI_I2S_IT 指定待检查的 SPI 中断源，包括：

- SPI_IT_OVR　　超出中断标志位
- SPI_IT_MODF　　模式错误标志位
- SPI_IT_CRCERR　　CRC 错误标志位
- SPI_IT_TXE　　发送缓存空中断标志位
- SPI_IT_RXNE　　接受缓存非空中断标志位

返回值为 SPI_I2S_IT 的新状态，值为 SET 或 RESET。

示例：

```
ITStatus Status;
Status = SPI_I2S_GetITStatus(SPI1, SPI_IT_OVR);
```

14）函数 SPI_I2S_ClearITPendingBit

功能描述：清除 SPIx 的中断待处理位。

函数原型：void SPI_I2S_ClearITPendingBit(SPI_TypeDef * SPIx，uint8_t SPI_I2S_IT)。

输入参数 SPIx 用于选择 SPI 外设，SPI_I2S_IT 为待清除的 SPI 中断源，与 SPI_I2S_Get_ITStatus 函数的第二个参数取值相同，中断标志位 BSY，TXE 和 RXNE 由硬件重置。

示例：

```
SPI_I2S_ClearITPendingBit(SPI2, SPI_IT_CRCERR);
```

10.6 SPI 案例

10.6.1 SPI 寄存器操作案例

1）SPI_Iint 函数实现

```
void SPI_Init(SPI_TypeDef * SPIx, SPI_InitTypeDef * SPI_InitStruct)
{
  uint16_t tmpreg = 0;
  //CR1 寄存器配置
  tmpreg = SPIx->CR1;
  //清除 BIDIMode, BIDIOE, RxONLY, SSM, SSI, LSBFirst, BR, MSTR, CPOL 和 CPHA
  tmpreg &= CR1_CLEAR_MASK;
  //根据 SPI_Direction 配置 BIDImode, BIDIOE 和 RxONLY 字段
  //根据 SPI_Mode 和 SPI_NSS 设置 SSM, SSI 和 MSTR 字段
  //根据 SPI_FirstBit 设置 LSBFirst
  //根据 SPI_BaudRatePrescaler 设置 BR 字段
  //根据 SPI_CPOL 和 SPI_CPHA 设置 CPOL 和 CPHA 字段
  tmpreg |= (uint16_t)((uint32_t)SPI_InitStruct->SPI_Direction
        |SPI_InitStruct->SPI_Mode | SPI_InitStruct->SPI_DataSize
        |SPI_InitStruct->SPI_CPOL | SPI_InitStruct->SPI_CPHA
        |SPI_InitStruct->SPI_NSS  | SPI_InitStruct->SPI_BaudRatePrescaler
        |SPI_InitStruct->SPI_FirstBit);
  SPIx->CR1 = tmpreg;
  //配置 CRC 多项式寄存器
  SPIx->CRCPR = SPI_InitStruct->SPI_CRCPolynomial;
}
```

2）中断状态读取函数实现

```
ITStatus SPI_I2S_GetITStatus(SPI_TypeDef * SPIx, uint8_t SPI_I2S_IT)
{
  ITStatus bitstatus = RESET;
  uint16_t itpos = 0, itmask = 0, enablestatus = 0;
  //根据 SPI_I2S_IT 的定义获取状态寄存器对应的中断状态字段的位置
  //例如 TXE 中断定义为 0x71,7 表示中断控制 TXEIE 的位置,1 表示 SR 寄存器 TXE 标志的位置
  itpos = 0x01 << (SPI_I2S_IT & 0x0F);
  //根据输入的中断源计算中断开关的字段位置
  itmask = SPI_I2S_IT >> 4;
  itmask = 0x01 << itmask;
  //判断中断源所对应的中断开关字段是否被打开
```

```
  enablestatus = (SPIx->CR2 & itmask) ;
  //如果允许中断且状态寄存器对应的状态位为 1,则返回 SET,否则返回 RESET
  if (((SPIx->SR & itpos) != (uint16_t)RESET) && enablestatus)
    bitstatus = SET;
  else
    bitstatus = RESET;
  return bitstatus;
}
```

10.6.2 SPI 函数库案例

1) 基本配置流程

- 调用 RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1, ENABLE)或 RCC_APB1Periph ClockCmd(RCC_APB1Periph_SPI2, ENABLE)使能 SPI1 和 SPI2 的时钟。
- 调用 RCC_AHBPeriphClockCmd()使能 SCK, MOSI, MISO,NSS 引脚对应的 GPIO 端口时钟。
- 调用 GPIO_PinAFConfig()将 I/O 与复选功能进行映射。
- 配置 GPIO 引脚为复选功能,上拉或下拉输出,调用 GPIO_Init 进行初始化。
- 配置极性、相位、波特率分频,主从模式等,通过 SPI_Init()进行初始化。
- 配置 NVIC,使能相应中断,调用 SPI_ITConfig()开启 SPI 中断源。
- 调用 SPI_Cmd()启用 SPI。
- 也可以通过调用 SPI_BiDirectionalLineConfig()配置三线模式, SPI_NSSInternalSoftwareConfig()配置 NSS 引脚,SPI_DataSizeConfig()配置数据位宽以及调用 SPI_SSOutputCmd()设置 NSS 的输入输出状态。

2) 库函数配置案例

```
void SPI_INIT()
{
  SPI_InitTypeDef SPI_InitStructure;
  GPIO_InitTypeDef GPIO_InitStructure;
  SPI_Cmd(SPI2,DISABLE);
  RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOB, ENABLE);          //使能 GPIOB 时钟
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI2, ENABLE);
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_15;                    // 初始化指定引脚
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;
  GPIO_Init(GPIOB, &GPIO_InitStructure);
```

```
    GPIO_PinAFConfig(GPIOB, GPIO_PinSource15, GPIO_AF_SPI2);
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_14;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure);
    GPIO_PinAFConfig(GPIOB, GPIO_PinSource14, GPIO_AF_SPI2);
    SPI_InitStructure.SPI_Direction= SPI_Direction_2Lines_FullDuplex;//全双工
    SPI_InitStructure.SPI_Mode= SPI_Mode_Master;          //SPI 主设备
    SPI_InitStructure.SPI_DataSize= SPI_DataSize_8b;      //SPI 传输数据 8 位
    SPI_InitStructure.SPI_CPOL= SPI_CPOL_Low;             //时钟极性为低跳变到高采集数据
    SPI_InitStructure.SPI_CPHA= SPI_CPHA_1Edge;           //时钟相位(第一个跳变沿采集数据)
    SPI_InitStructure.SPI_NSS= SPI_NSS_Soft;              //软件片选方式
    SPI_InitStructure.SPI_BaudRatePrescaler= SPI_BaudRatePrescaler_16;//1 分频
    SPI_InitStructure.SPI_FirstBit= SPI_FirstBit_MSB;     //数据传输从 MSB 位开始
    SPI_InitStructure.SPI_CRCPolynomial= 7;               //设置 crc 多项式
    SPI_Init(SPI2,&SPI_InitStructure);
    SPI_SSOutputCmd(SPI2, ENABLE);                        //主模式,NSS 引脚输出
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;
    GPIO_InitStructure.GPIO_Mode= GPIO_OType_PP;
    GPIO_Init(GPIOB, &GPIO_InitStructure);
    SPI_Cmd(SPI2,ENABLE);
}
```

数据发送

```
while(1)
{
    while(SPI_I2S_GetFlagStatus(SPI2,SPI_I2S_FLAG_BSY )!=RESET);
    SPI_I2S_SendData(SPI2, 0x01);
    while(SPI_I2S_GetFlagStatus(SPI2,SPI_I2S_FLAG_BSY )!=RESET);
}
```

10.6.3 温度传感器 ADT7320 案例

和 I2C 一样,对于特定的外设,其数据的读写有特定的字节传输时序要求,ADT7320 是一款 SPI 接口的温度传感器,其结构如图 10-20 所示。其中,SCLK、DOUT、DIN 和 CS 为 SPI 总线接口,DIN 为 MOSI,DOUT 为 MISO。

ADT7320 内部有多个寄存器,用于存储温度值和对传感器芯片进行配置,因此在对传感器操作时,我们需要通过 SPI 总线向 ADT7320 发送一个命令字,命令字中定义了对寄存器的读写操作方向和所要访问的寄存器地址,在主设备向 ADT7320 发送命令字时,主设备收到的数据无效,命令字格式如图 10-21 所示。

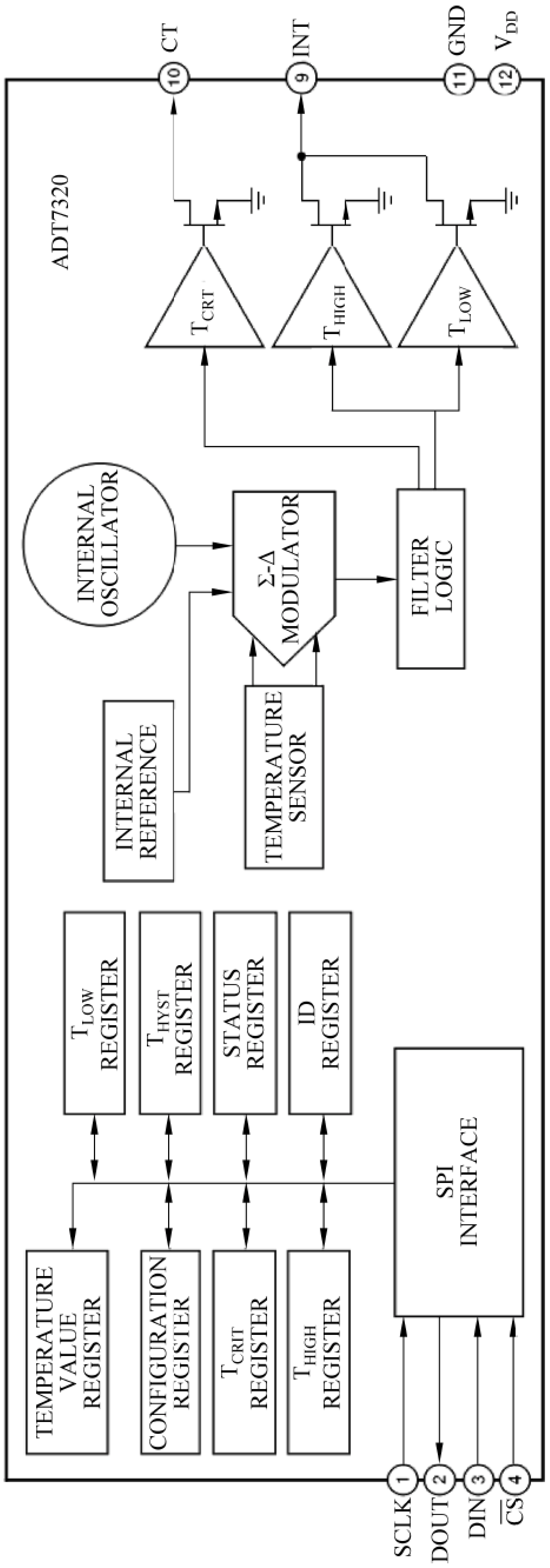


图 10-20　ADT7320 传感器芯片结构

C7	C6	C5	C4	C3	C2	C1	C0
0	R/$\overline{W}$	寄存器地址			0	0	0

图 10-21　命令字格式

ADT7320 向寄存器写一个字节的数据时序如图 10-22 所示，主设备首先发送命令字，命令字中 C6 置为 0，C5～C3 给出所要写的寄存器地址，然后紧接着主设备再发送一个字节的数据，从设备 ADT7320 发回的数据忽略。

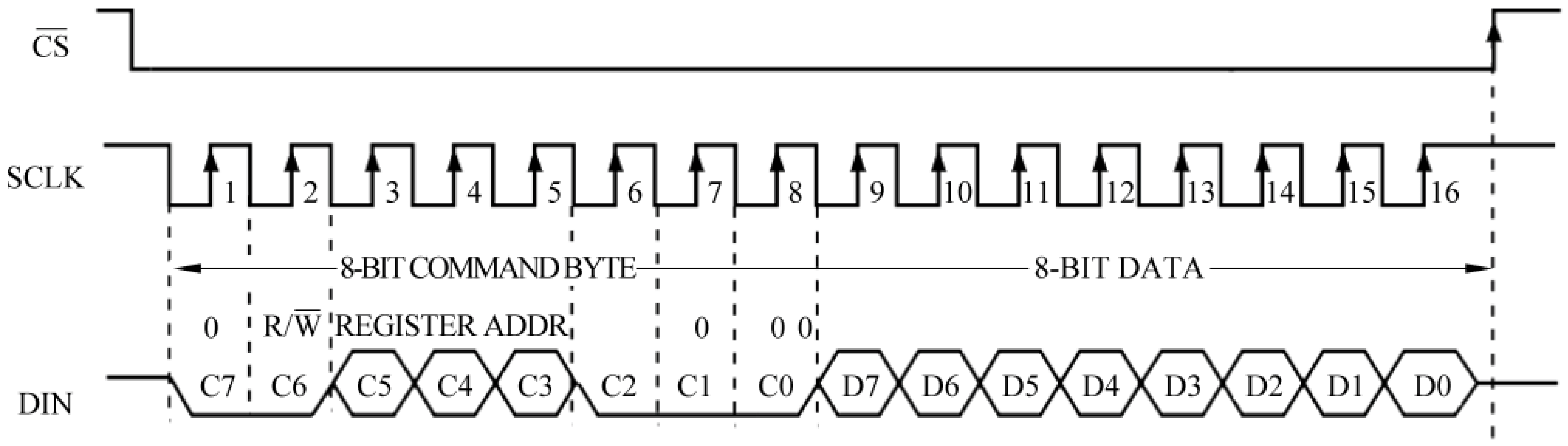


图 10-22　SPI 写寄存器时序

读寄存器值时，首先需要向 ADT7320 发送一个命令字，C6 置为 0，从机 ADT7320 的返回数据无效，然后主设备再向从设备发送一个任意数据，从设备将之前收到的命令字中寄存器地址所对应的数据发送到主设备，完成一次寄存器读操作，时序如图 10-23 所示。

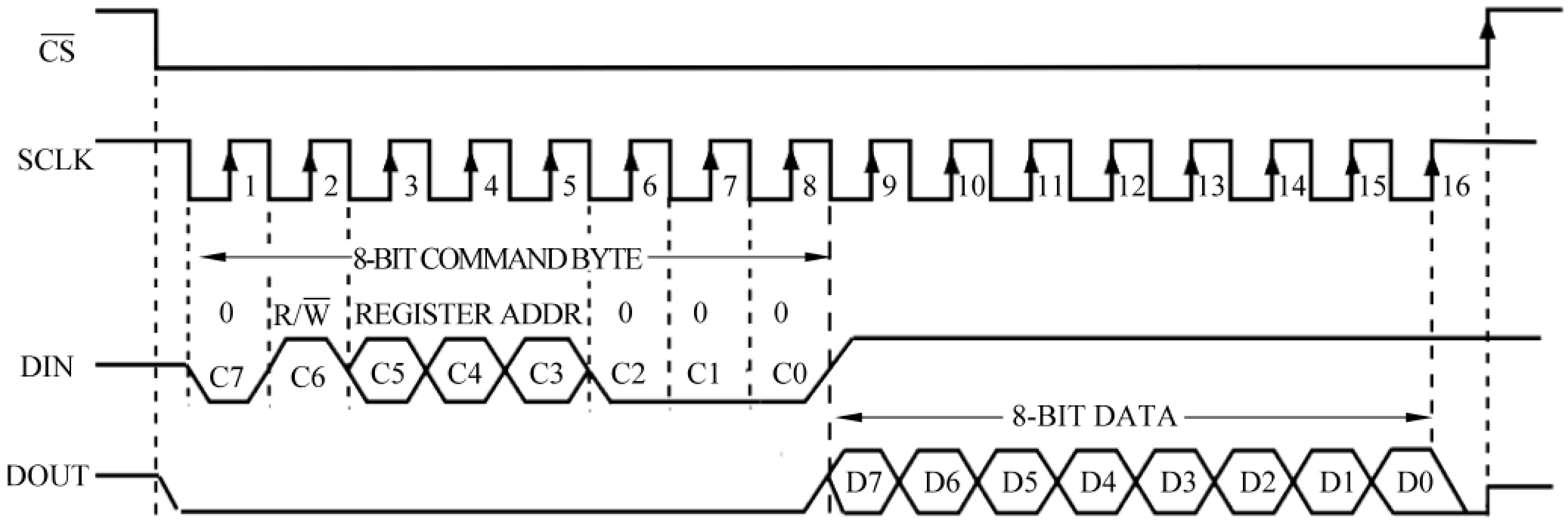


图 10-23　读寄存器时序

一个 I/O 模拟的读温度数值的函数实现如下：

```
unsigned int ReadFromADT7320ViaSPI(unsigned int reg_address)
{
    unsigned int   i;
    unsigned int spi_misoValue;
    unsigned int spi_Value;
```

```
    spi_Value = ( 0x78 & ((reg_address + 8) << 3));    //命令字
    OutputBit(CS,1);
    OutputBit(SCL,1);
    OutputBit(CS,0);
    Delay(5);
    for(i=0;i<8;i++)                                    //发送命令字
    {
        OutputBit(SCL,0);
        if((spi_Value & 0x80)==0x80)
            OutputBit(DIN,1);
        else
            OutputBit(DIN,0);
        Delay(5);
        OutputBit(SCL,1);
        spi_Value = (spi_Value <<1);
        Delay(5);
    }
    //读取寄存器的值
    for(i=0;i<8;i++)
    {
        OutputBit(SCL,0);
        spi_misoValue = (spi_misoValue <<1);
        Delay(10);
        if(InputBit(DOUT) ==1)
            spi_misoValue |=0x0001;
        else
            spi_misoValue &=0xfffe;
        Delay(2);
        OutputBit(SCL,1);
        Delay(8);
    }
    OutputBit(SCL,1);
    OutputBit(CS,1);
    return  spi_misoValue;
}
```

Chapter 11

第 11 章　模拟/数字转换

【导读】 在数据采集系统中，模数转换是至关重要的环节，在精度要求不高的情况下通常采用嵌入式微控制集成的模拟数字器实现模数转换。本章首先介绍模拟数字器采集的相关概念，然后对 STM32L152 片内集成 AD 的结构、寄存器和库函数使用方法进行了介绍，并对模拟数字器基本初始化和使用进行了案例阐述。

11.1　ADC 简介

1. 模拟信号和数字信号

模拟信号指信息参数在给定范围内表现为连续的信号，或在一段连续的时间间隔内，其代表信息的特征量可以在任意瞬间呈现为任意数值的信号，例如电压、电流与声音等。

数字信号指幅度的取值是离散的，幅值表示被限制在有限个数值之内。二进制码就是一种数字信号。二进制码受噪声的影响小，易于有数字电路进行处理，所以得到了广泛的应用。

2. 模拟数字转换器

模拟数字转换器(Analog-to-digital converter，ADC)是用于将模拟形式的连续信号转换为数字形式的离散信号的器件。典型的模拟数字转换器将模拟信号转换为表示一定比例电压值的数字信号。

通常的模数转换器是把经过与标准量比较处理后的模拟量转换成以二进制数值表示的离散信号的转换器，其工作原理如图 11-1 所示。输入的模拟信号 V(t)在采样时钟 CP 的控制下将连续量转变成了离散量，两个离散的采样点之间的信号由取样保持电路负责保持前

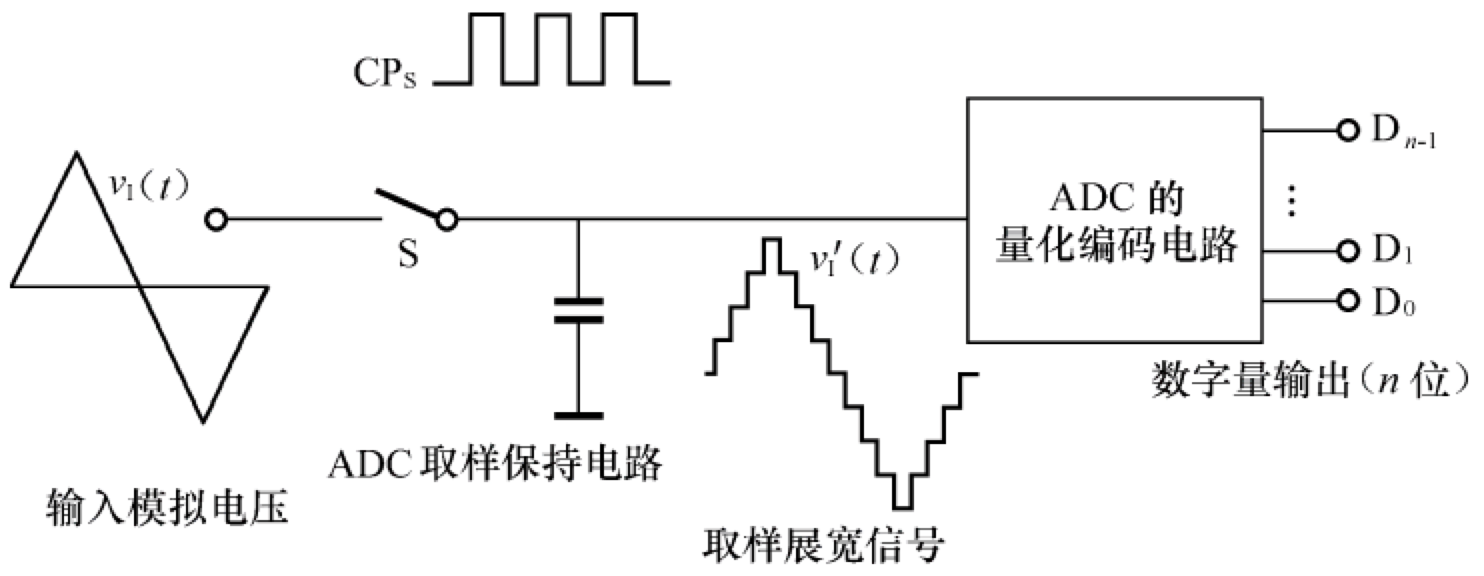


图 11-1　ADC 采样原理

一个采样点的电压。这样将“光滑”的模拟信号变成了“齿状”的取样展宽信号，对该信号和参考信号进行比较，衡量其大小，即可输出二进制编码，类似于十进制到二进制的转换。故任何一个模数转换器都需要一个参考模拟量作为转换的标准，比较常见的参考标准为最大的可转换信号大小，而输出的数字量则表示输入信号相对于参考信号的大小。

3. AD 采样过程

A/D 转换过程是通过取样、保持、量化和编码这四个步骤完成的采样样和保持主要由采样保持器来完成，量化编码由 A/D 转换器完成。

1）采样

采样是对模拟信号进行周期性抽样取值的过程，采样将连续的模拟信号分解成许多个离散点。如图 11-2 所示，采样开关在采样时钟的控制下对输入信号进行采样，为了保证采样后的信号能恢复模拟信号的特征，采样的时钟必须满足奈奎斯特采样定律，即采样频率应不小于输入模拟信号频谱中最高频率的两倍：Fsample >= 2 * Fmax_signal。

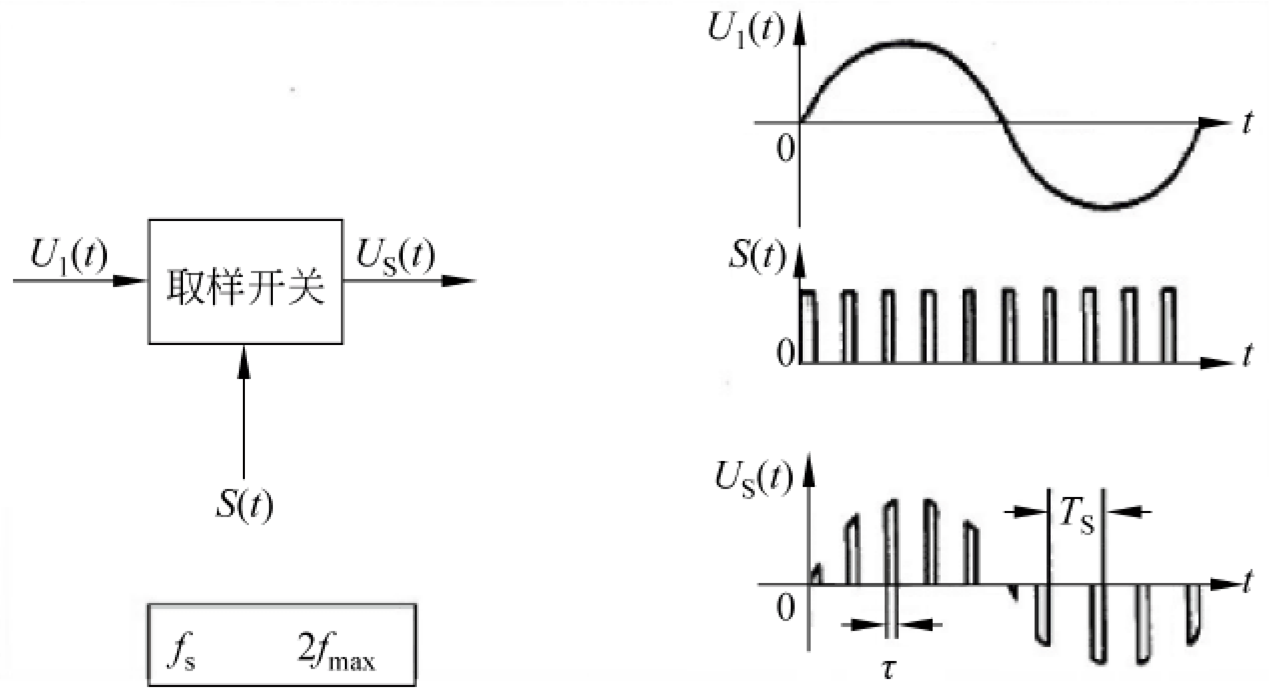


图 11-2 采样过程

2）保持

保持，即将采样点的电压信号保持一段时间。因为后续的量化过程需要一定的时间 τ，对于随时间变化的模拟输入信号，要求瞬时采样值在时间 τ 内保持不变，这样才能保证转换的正确性和转换精度，这个过程就是保持。如图 11-3 所示，采样保持后，原来连续模拟信号变成了阶梯形的连续信号。

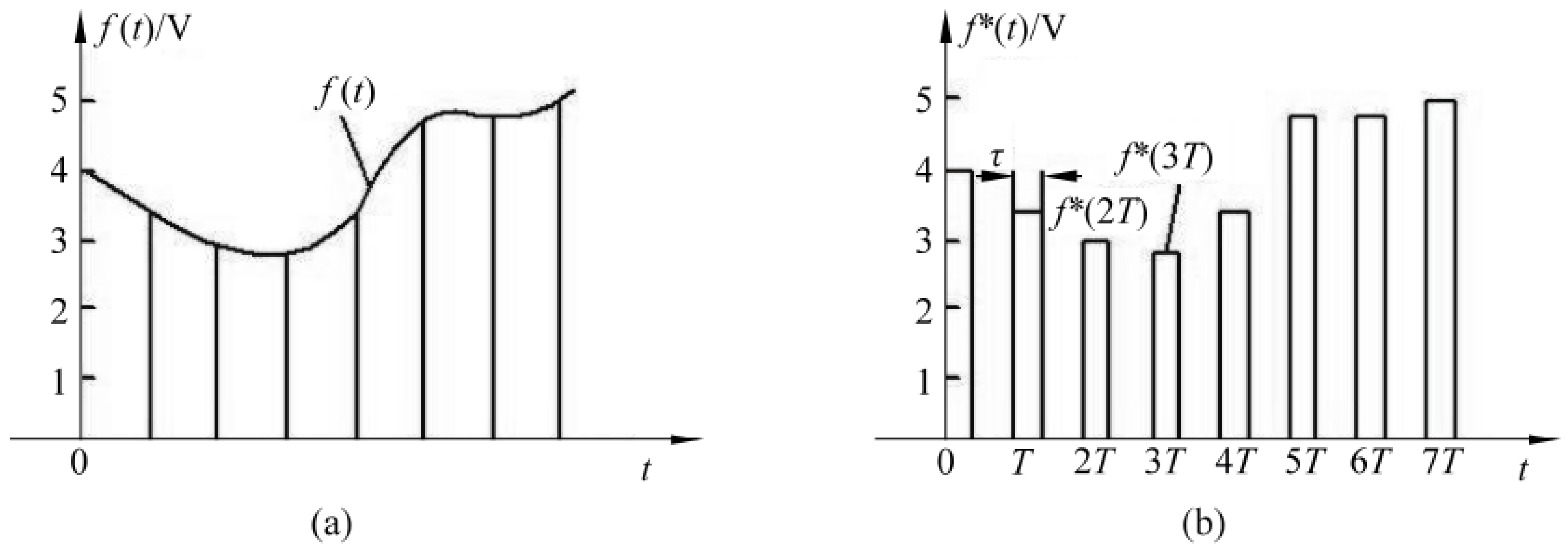


图 11-3 采样保持过程

3）量化

量化是对输入信号的幅值按照量化单位进行离散处理的过程。即采样将时间轴进行了离散化，量化将采样信号的幅值经过舍入或截尾的方法进行离散化，变为只有有限个有效数字的数。若取信号可能出现的最大值 A，令其分为 D 个间隔，则每个间隔长度为 R＝A/D，R 称为量化增量或量化步长，当采样信号落在某一小间隔内，经过舍入或截尾方法而变为有限值时，则产生量化误差，ADC 的量化位数一般采用 8，12，16，24 位，量化位数越大，分辨率越高，量化误差越小。

4）编码

编码是将一系列模拟信号采样和量化后用数字信号给描述出来的过程，这个数字信号序列就是编码。

4. ADC 的性能指标

衡量一个 ADC 的性能指标包括：

1）分辨率

分辨率（Resolution）是指 ADC 转化器所能分辨的模拟信号的最小变化值，分辨率的高低取决于量化位数，n 位的量化下，数字量变化一个最小量时模拟信号的变化量为：Vmax ＊1/2^n。例如 8 位的 AD，可以描述 256 个刻度的精度（2 的 8 次方），在它测量一个 0～5V 电压信号时，它的分辨率是 5V 除以 256，0.02V，即最小单位 0.02V。

2）转换速率

转换速率（Conversion Rate）是完成一次从模拟转换到数字的 A/D 转换所需的时间，转换速率决定了采样频率的最大值，因此间接影响了输入模拟信号的信号频率的最大值。不同类型的 ADC 转换速率不同，积分型 ADC 一次转化在毫秒级、逐次比较型 ADC 一次转换在微秒级，并行比较型 ADC 可以达到纳秒级。

3）量程

量程是 A/D 转换的输入信号电压范围，一般为 0～5V、0～10V、－5～＋5V、－10～＋10V，可以根据具体的输入信号进行调整。

4）信号连接方式

A/D 转换的输入信号可以采用单端方式或差分方式。单端方式的信号共用一个模拟地，抗干扰能力较差，但能提供更多的输入通道；查分方式每个通道需要两个引脚，信号之间互不影响，可对差分信号进行采集，抗干扰能力强。

5）量化误差

量化误差（Quadratuer Error）是指由于对模拟信号进行量化而产生的误差，量化结果和被量化模拟量的差值，显然量化位数越多，量化的相对误差越小，A/D 的量化误差为 1 或 1/2 的最小量化步长（取决于量化时的舍入方式是截断还是四舍五入），如图 11-4 所示，最差情况下是一个量化步长，最好情况是 1/2 量化步长。

6）偏移误差

偏移误差（Offset Error）是指输入信号为零时输出信号不为零引起的误差值，一般以满量程电压值的百分比表示。大多数的偏移误差是由温度引起的，可以通过外部电路进行调零。

7）满刻度误差

满刻度误差（Full Scale Error）也叫增益误差，满度输出时对应的输入信号与理想输入

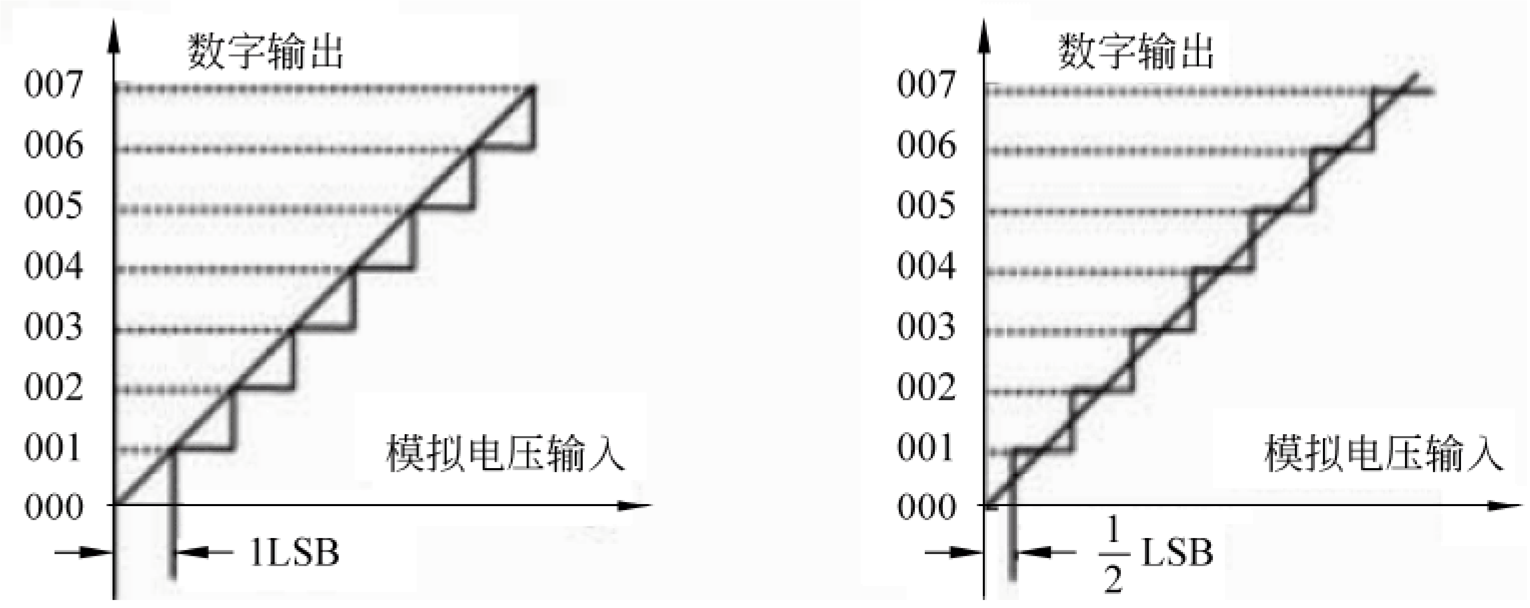


图 11-4　量化误差

信号值之差,也采用满量程电压值的百分比表示。

8）线性度

线性度(Linearity)指在没有偏移和增益误差的情况下,实际转换器的转移函数与理想直线的最大偏移。线性误差是由 ADC 特性随输入信号幅度变化引起的,因此不能补偿。

5. A/D 转换器类型

A/D 转换器主要分为积分型、逐次逼近型、并行比较型/串并行型、Σ-Δ 调制型、电容阵列逐次比较型及压频变换型等。

1）积分型

积分型 A/D 工作原理是将输入电压转换成时间(脉冲宽度信号)或频率(脉冲频率),然后由定时器/计数器获得数字值。其优点是用简单电路就能获得高分辨率,但缺点是由于转换精度依赖于积分时间,因此转换速率极低。初期的单片 A/D 转换器大多采用积分型,现在逐次比较型已逐步成为主流。

2）逐次逼近型

逐次逼近型 A/D 由一个比较器和 D/A 转换器通过逐次比较逻辑构成,从 MSB 开始,顺序地对每一位将输入电压与内置 D/A 转换器输出进行比较,经 n 次比较而输出数字值。其电路规模属于中等。其优点是速度较高、功耗低,在低分辨率(＜12 位)时价格便宜,但高精度(＞12 位)时价格很高。

3）并行比较型/串并行比较型

并行比较型 A/D 采用多个比较器,仅作一次比较而实行转换,又称 Flash(快速)型。由于转换速率极高,n 位的转换需要 $2n-1$ 个比较器,因此电路规模也极大,价格也高,只适用于视频 A/D 转换器等速度特别高的领域。

串并行比较型 A/D 结构上介于并行比较型和逐次逼近型之间,最典型的是由 2 个 $n/2$ 位的并行型 A/D 转换器配合 D/A 转换器组成,用两次比较实行转换,所以称为 Half flash(半快速)型。还有分成三步或多步实现 A/D 转换的叫做分级(Multistep/Subrangling)型 A/D,而从转换时序角度又可称为流水线型 A/D,现代的分级型 A/D 中还加入了对多次转换结果作数字运算而修正特性等功能。这类 A/D 速度比逐次比较型高,电路规模比并行型小。

4）Σ-Δ 调制型

Σ-Δ 型 A/D 由积分器、比较器、1 位 DA 转换器和数字滤波器等组成。原理上近似于积

分型，将输入电压转换成时间（脉冲宽度）信号，用数字滤波器处理后得到数字值。电路的数字部分基本上容易单片化，因此容易做到高分辨率。主要用于音频和测量。

5）电容阵列逐次比较型

电容阵列逐次比较型 A/D 在内置 D/A 转换器中采用电容矩阵方式，也可称为电荷再分配型。一般的电阻阵列 D/A 转换器中多数电阻的值必须一致，在单芯片上生成高精度的电阻并不容易。如果用电容阵列取代电阻阵列，可以用低廉成本制成高精度单片 A/D 转换器。最近的逐次比较型 A/D 转换器大多为电容阵列式的。

6）压频变换型

压频变换型（Voltage-Frequency Converter）是通过间接转换方式实现模数转换的。其原理是首先将输入的模拟信号转换成频率，然后用计数器将频率转换成数字量。从理论上讲这种 A/D 的分辨率几乎可以无限增加，只要采样的时间能够满足输出频率分辨率要求的累积脉冲个数的宽度。其优点是分辨率高、功耗低、价格低，但是需要外部计数电路共同完成 A/D 转换。

6. 逐次逼近型 ADC 的工作原理

图 11-5 为逐次逼近型的结构图。这种 A/D 转换器是以 D/A 转换器为基础，加上比较器、逐次逼近寄存器、置数选择逻辑电路及时钟等组成。

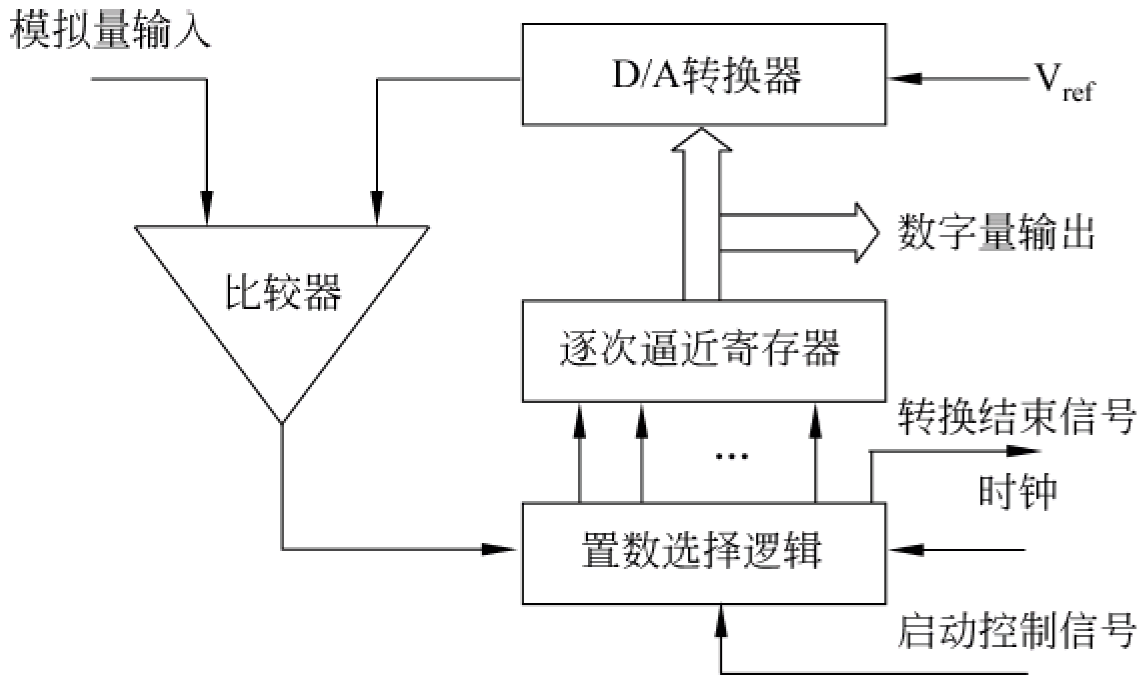


图 11-5　逐次逼近型电路结构图

其转换原理如图 11-6 所示。在启动信号控制下，首先置数选择逻辑电路，给逐次逼近寄存器最高位置 1，经 D/A 转换成模拟量后与输入模拟量进行比较，电压比较器给出比较结果。如果输入量大于或等于经 D/A 变换后输出的量，则比较器为 1，否则为 0，置数选择逻辑电路根据比较器输出的结果，修改逐次逼近寄存器中的内容，使其经 D/A 变换后的模拟量逐次逼近输入模拟量。这样经过若干次修改后的数字量，便是 A/D 转换结果的量。

逼近型 A/D 大多采用二分搜索法，即首先取允许电压最大范围的 1/2 值与输入电压值进行比较，也就是首先最高为 1，其余位为 0。如果搜索值在此范围内，则再取范围的 1/2 值，即次高位置 1。如果搜索值不在此范围内，则应以搜索值的最大允许输入电压值的另外 1/2 范围，即最高位为 0，依次进行下去，每次比较将搜索范围缩小 1/2，具有 n 位的 A/D 变换，经 n 次比较，即可得到结果。因此，必须在 A/D 转换结束后才能从逐次逼近寄存器中取出数字量。为此 D/A 芯片专门设置了转换结束信号引脚，向 CPU 发转换结束信号，通知 CPU 读取转换

后的数字量,CPU 可以通过中断或查询方式检测 A/D 转换结束信号,并从 A/D 芯片的数据寄存器(即图 10-9 中逐次逼近寄存器)中取出数字量。逐次逼近法变换速度较快,所以集成化的 A/D 芯片多采用上述方法,STM32L152 内部集成的 ADC 便是逐次逼近型 ADC。

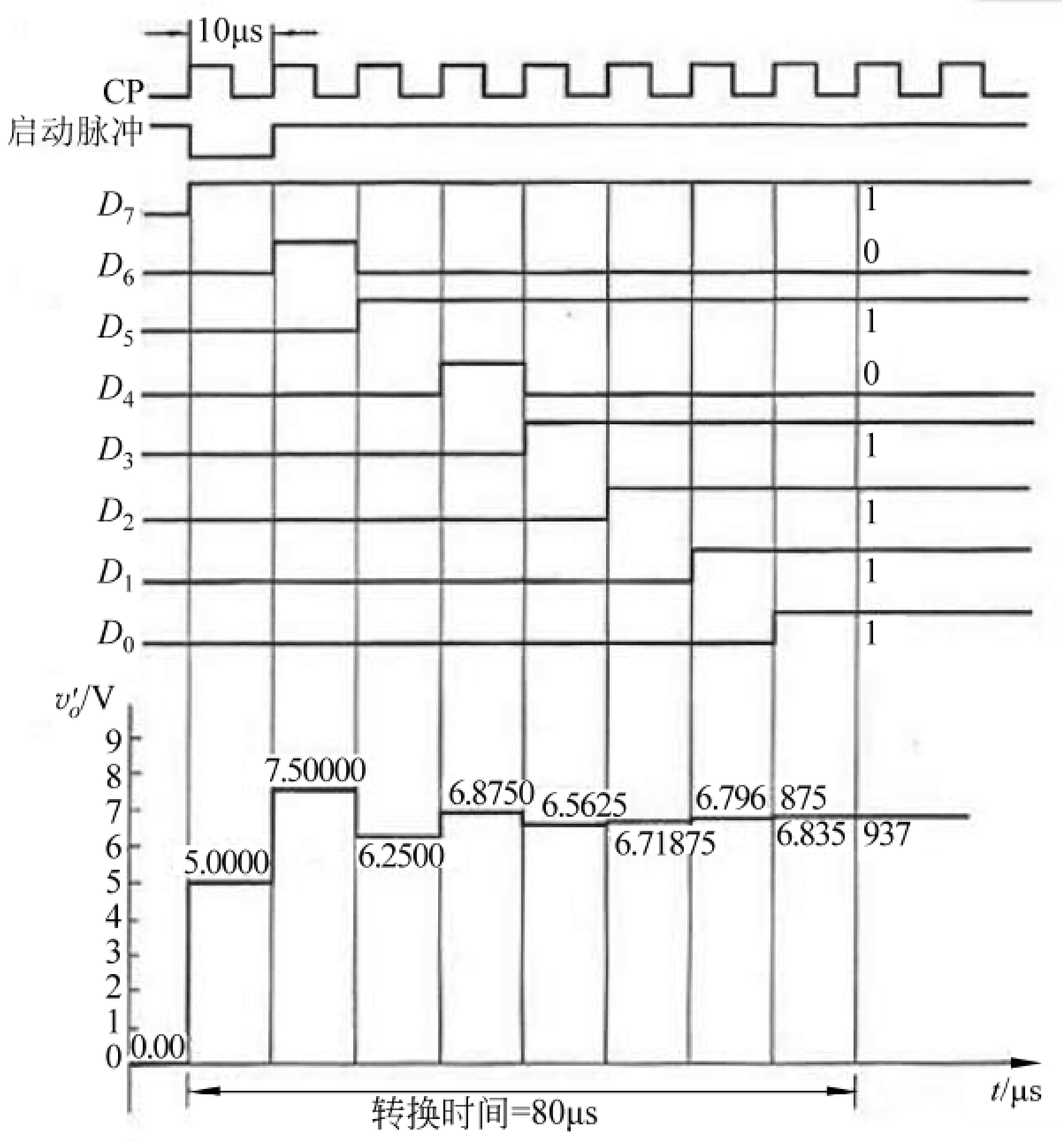


图 11-6　逐次逼近型 ADC 转换原理

11.2　STM32L152 ADC

STM32 内部集成了一个 12 位的逐次逼近型模拟数字转换器,最多支持 42 个通道,可以测量 40 个外部引脚的信号和 2 个内部信号,每个通道支持单次、连续、扫描或间断模式,转换结果以左对齐或右对齐方式存储在 16 位数据寄存器中。在给定的系统时钟驱动下,ADC 转换默认以最快速度执行,同时支持动态电源管理,旨在 ADC 转换期间供电,以降低功耗,其主要特征为:

- 12 位、10 位、8 位、6 位可配置的量化位数。
- 转换结束、注入转换结束和发生模拟看门狗事件、溢出事件时产生中断。
- 支持单次和连续转换模式。
- 支持可编程通道次序的自动扫描模式。
- 转换结果支持左对齐或右对齐方式。
- 每个 ADC 通道支持单独的采样时间配置。
- 规则通道和注入通道均有外部触发选项。

- ADC 转换时间：最大转换速率 1μs(ADCCLK＝16MHz)到 4μs(ADCCLK＝4MHz)。
- 可设置的自动关机模式以降低功耗。
- ADC 供电要求：高速 2.4～3.6V，低速 1.8V。
- ADC 输入范围：VREF－≤VIN≤VREF＋。

其内部结构如图 11-7 所示。

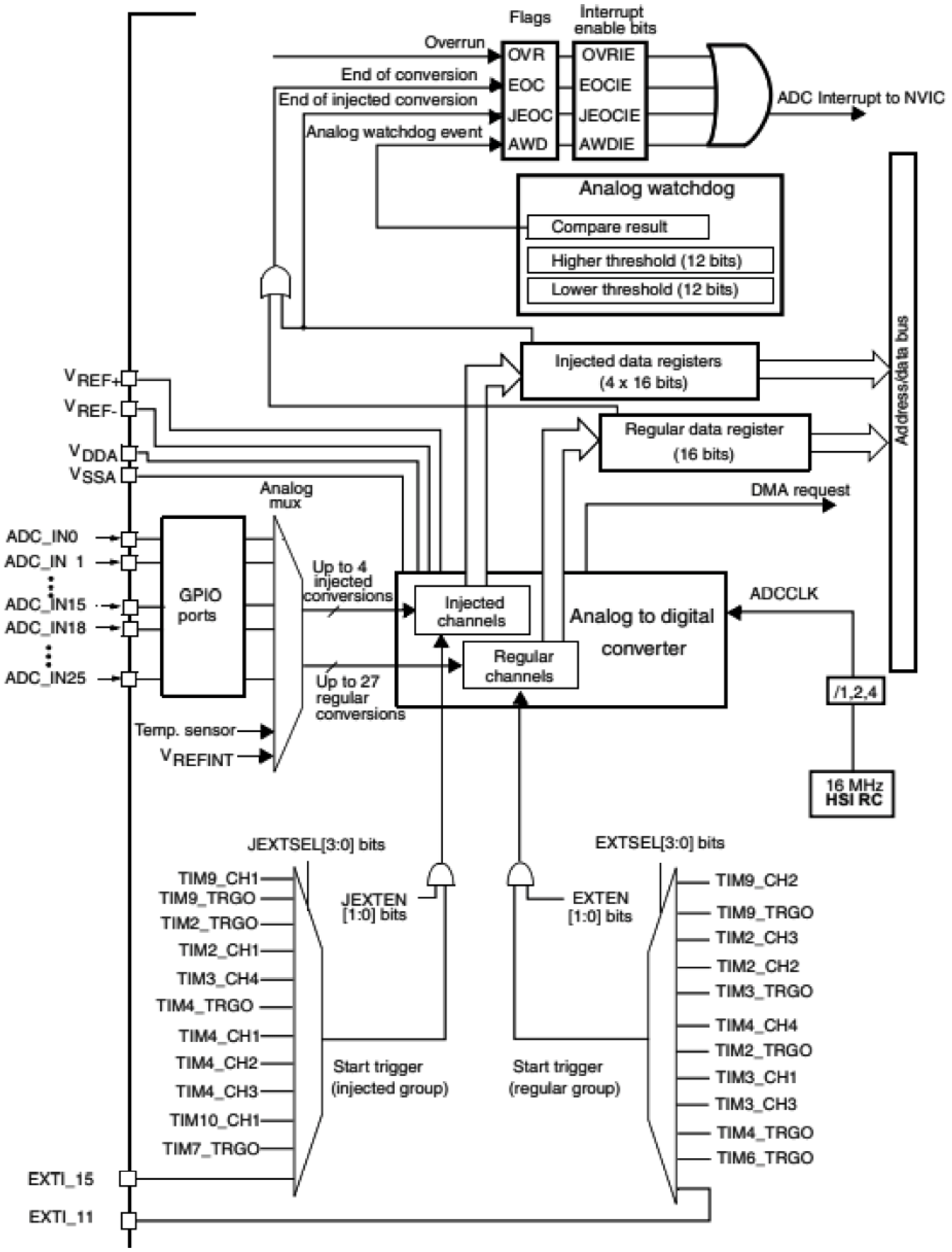


图 11-7　ADC 内部结构图

ADC 对外的接口见表 11-1 所示。一般情况下，VDD 小于 3.6V，VSS 接地，相对应的，VDDA 小于 3.6V，VSSA 也接地，模拟输入信号不要超过 VDD。

表 11-1　ADC 接口定义

名称	信 号 类 型	注　　解
VREF＋	输入，模拟参考正极	ADC 使用的参考电压，全速 ADCCLK＝16MHz，VREF＋ ＝VDDA≥2.4V；中速 ADCCLK＝8MHz 要求 VREF＋ ＝VDDA≥1.8V，若 VREF＋≠VDDA，则 VREF＋需大于 2.4V；低速 ADCCLK＝4MHz，VREF＋需大于 1.8V
VDDA	输入，模拟电源	等效于 VDD 的模拟供电电源， 全速要求 2.4V≤VDDA≤VDD(3.6V)， 中低速要求 1.8V≤VDDA≤VDD(3.6V)
VREF－	输入，模拟参考负极	ADC 使用的负极参考电压，VREF－＝VSSA
VSSA	输入，模拟电源地	等效于 VSS 的模拟电源地
ADC_INx	模拟输入信号	21～40 通道模拟输入通道

STM32L152 内部集成了 1 个 ADC，支持 21 个外部通道，可以作为 ADC 输入的 GPIO 引脚如表 11-2 所示。

表 11-2　STM32L152RET6 的 ADC 外部引脚

ADC_INx	GPIO	ADC_INx	GPIO
ADC_IN0	PA0	ADC_IN11	PC1
ADC_IN1	PA1	ADC_IN12	PC2
ADC_IN2	PA2	ADC_IN13	PC3
ADC_IN3	PA3	ADC_IN14	PC4
ADC_IN4	PA4	ADC_IN15	PC5
ADC_IN5	PA5	ADC_IN18	PB12
ADC_IN6	PA6	ADC_IN19	PB13
ADC_IN7	PA7	ADC_IN20	PB14
ADC_IN8	PB0	ADC_IN21	PB15
ADC_IN9	PB1	ADC_IN0b	PB2
ADC_IN10	PC0		

11.2.1　STM32L152 ADC 功能

STM32L52RET6 带 1 个 ADC 控制器，一共支持 23 个通道，包括 21 个外部和 2 个内

部信号源，内部信号源 ADC_IN16 和 ADC_IN17 分别被连接到了温度传感器和内部参照电压 VREFINT 上。

1. ADC 供电控制

通过设置 ADC_CR2 寄存器的 ADON 位可给 ADC 上电。当第一次设置 ADON 位时，它将 ADC 从断电状态下唤醒。通过清除 ADON 位可以停止转换，并将 ADC 置于断电模式。在这个模式中，ADC 几乎不耗电。ADC 上电后，上电延迟一段时间后（tSTAB）当 SWSTART、JSWSTART 位或者外部触发信号到达时，ADC 启动转换。

为降低功耗，当 ADC 转换就绪时（ADONS=1），ADC 自动对电源进行管理，通过 ADC_CR1 寄存器的 PDI 和 PDD 位，在 ADC 没有转换时自动进入掉电状态。PDI 用于配置 ADC 在等待软件或外部信号触发转换的状态是否自动掉电，PDD 用于表示两个转换之间的延迟内 ADC 是否自动掉电。

2. ADC 时钟配置

从图 11-8 可以看出，ADC 的模拟部分时钟（即采样时钟）来源独立于总线时钟，采用内部高速时钟 HSI，HSI 最高 16MHz，即无论 MCU 主频多少，ADC 的最高速率为 16MHz。

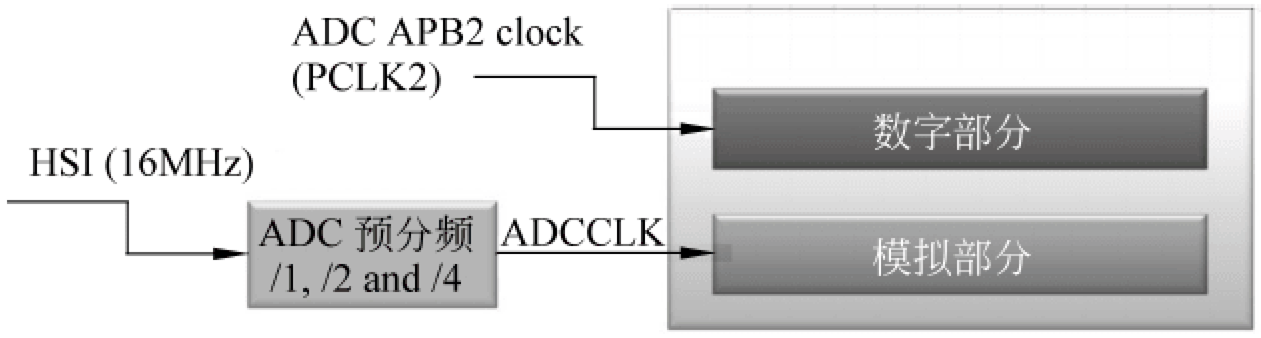


图 11-8　ADC 时钟

ADC 的工作时钟为 ADCCLK，ADCCLK 通过 ADC 预分频器可以对 HSI 时钟进行分频：

- 1 分频时为全速，ADCCLK=16MHz；
- 2 分频时为中速，ADCCLK=8MHz；
- 4 分频时为低速，ADCCLK=4MHz；

ADC 控制器挂接在 APB2 总线上，数字部分的数据交互需要使用总线时钟，由于 ADC 采样时钟可能会高于总线时钟，此时有可能引起 ADC 采集的数据无法及时被 CPU 取走，可以通过注入延迟等方法降低数据采集的速度。ADC 的总线接口通常由时钟控制器提供的 ADCCLK 时钟和 PCLK2（APB2 时钟）同步。RCC 控制器为 ADC 时钟提供一个专用的可编程预分频器。

3. ADC 通道选择

STM32 的 ADC 控制器有很多通道，所以模块通过内部的模拟多路开关，可以切换到不同的输入通道并进行转换。在任意多个通道上以任意顺序进行的一系列转换构成成组转换。例如，可以如下顺序完成转换：通道 3、通道 8、通道 2、通道 2、通道 0、通道 2、通道 2、通道 15。STM32 特别地加入了多种成组转换的模式，可以由程序设置好之后，对多个模拟通道自动地进行逐个地采样转换。它们可以组织成两组：规则通道组和注入通道组。

(1) 规则组由多达 28 个转换组成。规则通道和它们的转换顺序在 ADC_SQRx 寄存器中选择。规则组中转换的总数应写入 ADC_SQR1 寄存器的 L[3：0]位中。

(2) 注入组由多达 4 个转换组成。注入通道和它们的转换顺序在 ADC_JSQR 寄存器中选择。注入组里的转换总数目应写入 ADC_JSQR 寄存器的 L[1：0]位中。

如果 ADC_SQRx 或 ADC_JSQR 寄存器在转换期间被更改，当前的转换被清除，一个新的启动脉冲将发送到 ADC 以转换新选择的组。

规则组和注入组的关系如图 11-9 所示。规则通道组的转换是按照既定的序列正常执行，而注入通道组的转换则是打断规则组的执行优先执行的一组转换，类似于程序和中断服务程序。

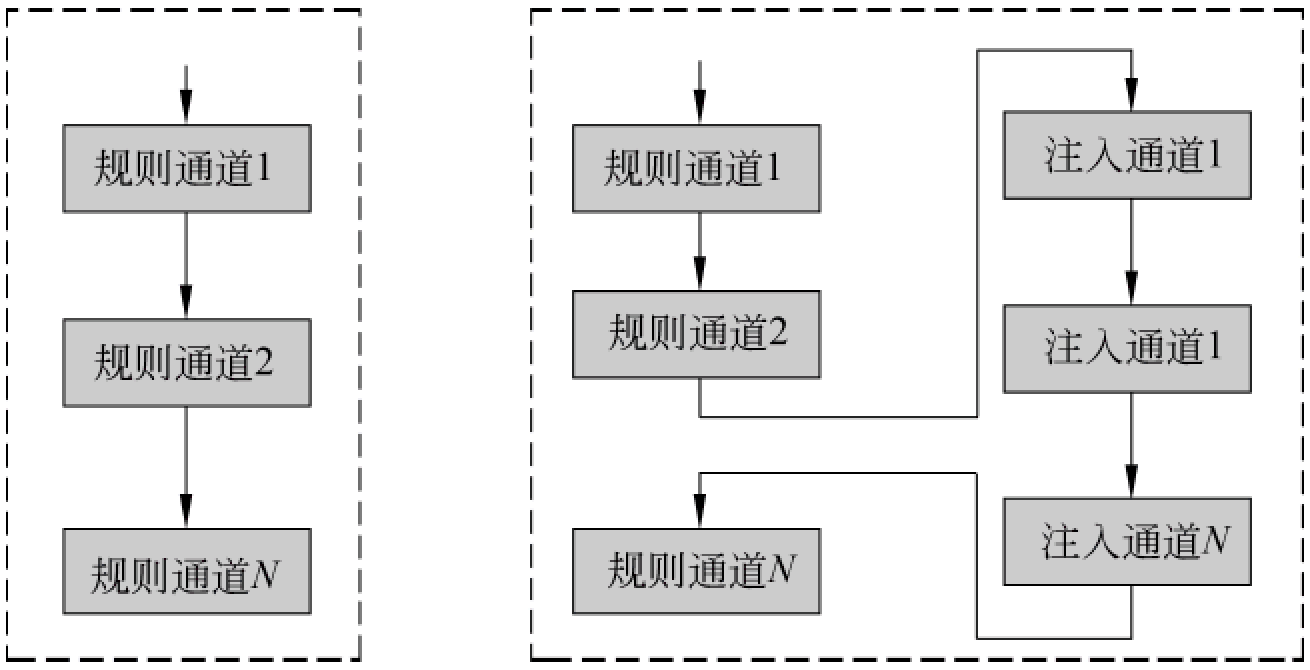


图 11-9　规则组和注入组的对比

4. 注入通道管理

注入通道可以通过软件或外部触发进行注入，也可以通过软件设置自动注入。

(1) 触发注入：使用触发注入时，必须清除 ADC_CR1 寄存器的 JAUTO 位，具体方式如下：

- 利用外部触发或通过设置 ADC_CR2 寄存器的 JSWSTART 位，启动一组注入通道的转换；
- 如果在规则通道转换期间产生一外部注入触发，当前转换被复位，注入通道序列被以单次扫描方式进行转换。
- 恢复上次被中断的规则组通道转换。

如果在注入转换期间产生一规则事件，注入转换不会被中断，但是规则序列将在注入序列结束后被执行。

(2) 自动注入：如果设置了 JAUTO 位，在规则组通道之后，注入组通道被自动转换。这可以用来转换在 ADC_SQRx 和 ADC_JSQR 寄存器中设置的最多 31 个转换序列。自动注入模式中，必须禁止注入通道的外部触发，如果除 JAUTO 位外还设置了 CONT 位，规则通道至注入通道的转换序列被连续执行。

5. ADC 转换方式

ADC 支持四种转换模式：单通道单次转换、单通道连续转换、多通道单次转换、多通道

连续转换，其关系如图 11-10 所示。

图 11-10　四种转换模式

1）单次转换模式

单次转换模式下，ADC 只执行一次转换。该模式下 ADC_CR2 寄存器的 CONT 为 0，当设置 ADC_CR2 寄存器的 SWSTART 位（规则通道）或 JSWSTART 位（注入通道）或者外部触发时（规则、注入通道均可）转换开始，转换完成后 ADC 停止。

• 如果一个规则通道被转换：

— 转换数据被储存在 16 位 ADC_DR 寄存器中；

— EOC（转换结束）标志被设置；

— 如果设置了 EOCIE，则产生中断。

• 如果一个注入通道被转换：

— 转换数据被储存在 16 位的 ADC_DRJ1 寄存器中；

— JEOC（注入转换结束）标志被设置；

— 如果设置了 JEOCIE 位，则产生中断。

2）连续转换模式

在连续转换模式中，当前面 ADC 转换一结束马上就启动另一次转换。此模式可通过外部触发启动或通过设置 ADC_CR2 寄存器上的 ADON 位启动，此时 CONT 位是 1。

• 如果一个规则通道被转换：

— 转换数据被储存在 16 位的 ADC_DR 寄存器中；

— EOC（转换结束）标志被设置；

— 如果设置了 EOCIE，则产生中断。

转换通道在 SQR5 寄存器的 SQ1[4：0]中指定。注入通道无法进行连续转换，只有注入通道被配置成规则通道之后的连续转换模式时（JAUTO=1）才能进行注入通道转换：

— 转换数据被储存在 16 位的 ADC_DRJ1 寄存器中

— JEOC（注入转换结束）标志被设置；

— 如果设置了 JEOCIE 位，则产生中断。

转换时序如图 11-11 所示。

3）扫描模式

扫描模式用于对一组模拟通道进行转换。扫描模式可通过设置 ADC_CR1 寄存器的

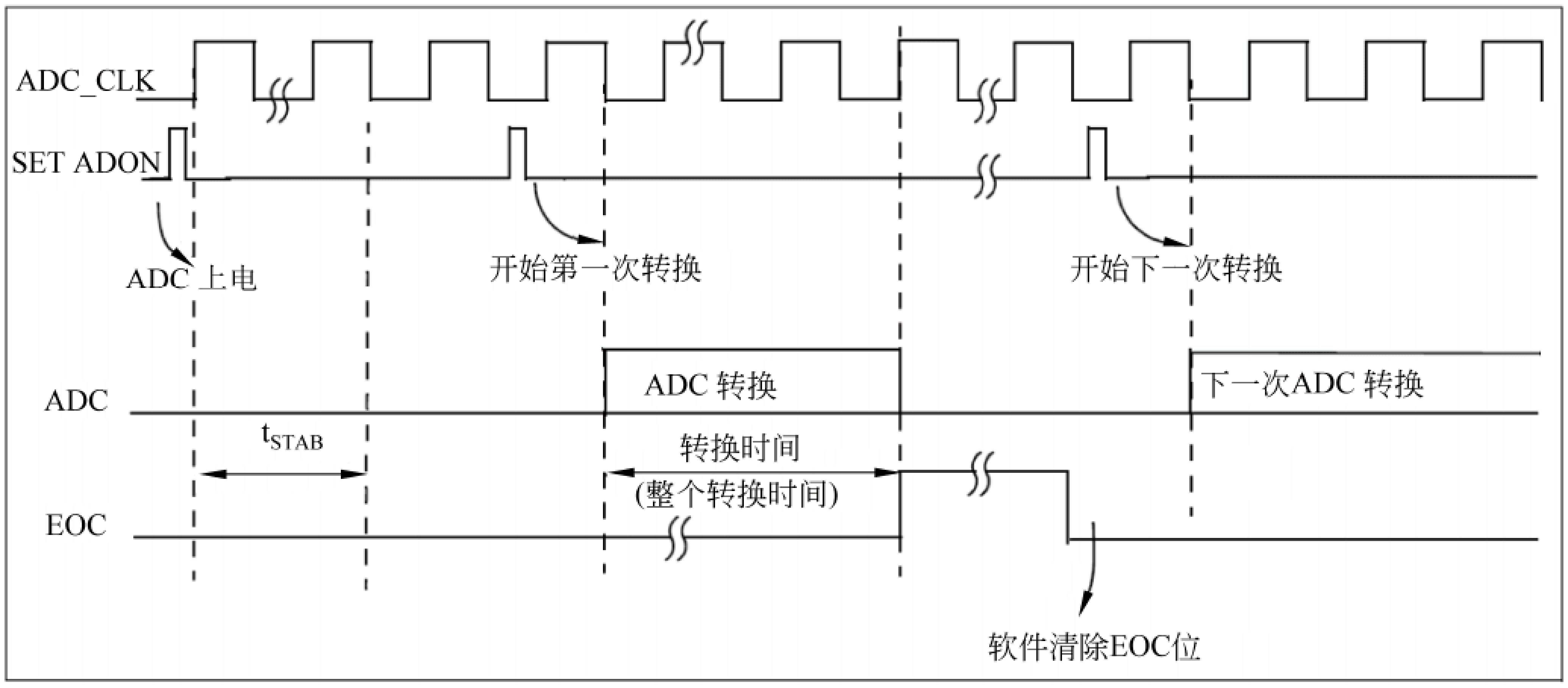


图 11-11　ADC 转换时序

SCAN 位来选择。一旦这个位被设置，ADC 扫描所有被 ADC_SQRX 寄存器（规则通道）或 ADC_JSQR（注入通道）选中的所有通道。在每个组的每个通道上执行单次转换。在每个转换结束时，同一组的下一个通道被自动转换。如果设置了 CONT＝1，则转换不会在选择组的最后一个通道上停止，而是再次从选择组的第一个通道继续转换，即多通道连续扫描模式，否则为多通道单次扫描模式。规则通道的每一组转换完成后状态寄存器 ADC_SR 的 EOC 清零，每个规则组的通道转换完后状态寄存器 ADC_SR 的 EOC 位置 1。如果在使用扫描模式的情况下使用中断，会在最后一个通道转换完毕后才会产生中断。而连续转换，是在每次转换后，都会产生中断。

4）间断模式

间断模式是一种特殊的扫描模式，对于规则组通道，此模式通过设置 ADC_CR1 寄存器上的 DISCEN 位激活。它可以用来执行一个子序列的 n 次转换（$n \leqslant 8$），此转换是 ADC_SQRx 寄存器所选择的转换序列的一部分。数值 n 由 ADC_CR1 寄存器的 DISCNUM[2：0]位给出。

一个外部触发信号可以启动 ADC_SQRx 寄存器中描述的下一轮 n 次转换，直到此序列所有的转换完成为止。所有的规则组的子序列转换完成后，下次开始从第一个子序列开始转换，总的序列长度由 ADC_SQR1 寄存器的 L[3：0]定义。例如：$n=3$，被转换的通道＝0、1、2、3、6、7、9、10。

- 第一次触发：转换的序列为 0、1、2；
- 第二次触发：转换的序列为 3、6、7；
- 第三次触发：转换的序列为 9、10，并产生 EOC 事件；
- 第四次触发：转换的序列 0、1、2。

对于注入组，此模式通过设置 ADC_CR1 寄存器的 JDISCEN 位激活。在一个外部触发事件后，该模式用于执行 ADC_JSQR 寄存器的一个子序列的 n 次转换（$n \leqslant 3$），n 由 ADC_

CR1 寄存器的 DISCNUM[2：0]位给出。

一个外部触发信号可以启动 ADC_JSQR 寄存器选择的下一个通道序列的转换，直到序列中所有的转换完成为止。总的序列长度由 ADC_JSQR 寄存器的 JL[1：0]位定义，例如 $n=1$，被转换的通道＝1、2、3。

- 第一次触发：通道 1 被转换；
- 第二次触发：通道 2 被转换；
- 第三次触发：通道 3 被转换，并且产生 EOC 和 JEOC 事件；
- 第四次触发：通道 1 被转换。

规则组转换的另一个例子如图 11-12 所示。ADC_SQR1 的值为：0、1、2、4、5、8、9、11、12、13、14、15；间隔转换的通道数量为 3。

图 11-12　规则通道转换案例

6. 数据对齐

ADC 转换的数据保存在数据寄存器，数据寄存器 16 位，但 ADC 最高支持到 12 位，因此数据的存储可以以左对齐或右对齐的方式，如图 11-13 和图 11-14 所示，ADC_CR2 寄存器中的 ALIGN 位选择转换后数据储存的对齐方式。

注入组

SEXT	SEXT	SEXT	SEXT	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

规则组

0	0	0	0	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

图 11-13　数据左对齐

注入组

SEXT	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	0	0	0

规则组

D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	0	0	0	0

图 11-14　数据右对齐

对于注入组通道，转换的数据值已经减去了用户在 ADC_JOFRx 寄存器中定义的偏移

量,因此结果可以是一个负值,SEXT 位是扩展的符号值。对于规则组通道,不需减去偏移值,因此只有 12 个位有效。

7. 通道采样时间

ADC 使用若干 ADC_CLK 周期对输入电压采样,采样周期数目可以通过 ADC_SMPRx(x=0、1、2)寄存器中的 SMP[2:0]位更改。每个通道可以分别用不同的时间采样。总的转换时间 TCONV=采样时间+转换时间。每个通道的转换时间由 SMP[2:0]决定,其取值为 4、9、16、24、48、96、192、384 倍的 ADCCLK 周期,如图 11-15 所示。

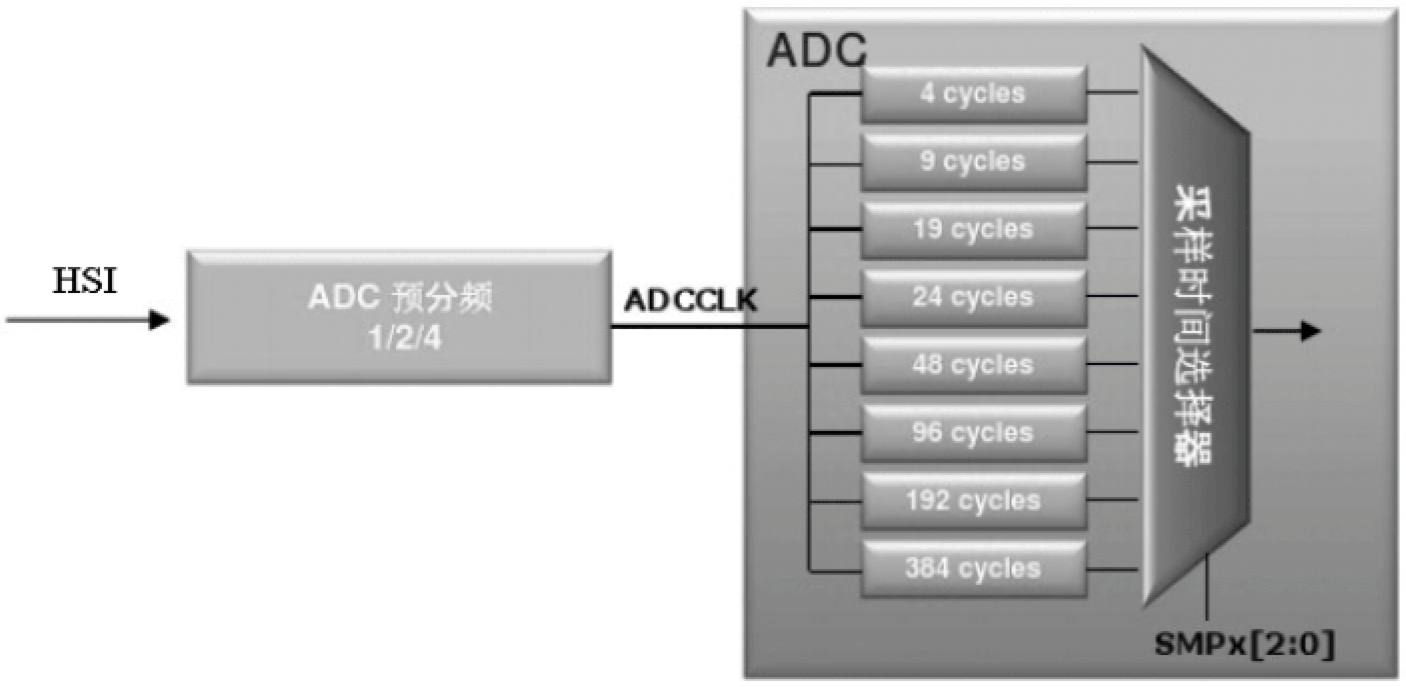


图 11-15　采样周期设置

转换周期与 ADC 的转换位数有关,如图 11-16 所示,当 ADC 分辨率为 12 位时,转换时间为 12 个 ADCCLK 周期,6 位的分辨率时转换时间最短可达 7 个 ADCCLK 周期。因此一个次 ADC 转换的时间例子如下:若 ADCCLK=16MHz,采样时间为 4 个周期,则:

- 12 位分辨率 Tconv = 4+12 = 16 周期=1μs。
- 10 位分辨率 Tconv = 4+11 = 15 周期=937.5ns。
- 8 位分辨率 Tconv = 4+9 = 13 周期=812.5ns。
- 6 位分辨率 Tconv = 4+7 = 11 周期=685ns。

分辨率	转换时间($T_{Conversion}$)
12 bit	12 Cycles
10 bit	11 Cycles
8 bit	9 Cycles
6 bit	7 Cycles

图 11-16　转换时间

这样,我们可以对 ADC 转换通道次序、采样时间和采样次数进行灵活的配置,例如,对 0、2、8、4、7、3、3、3 和 11 通道配置不同的采样时间进行转换,其中通道 3 进行 3 次采样,如图 11-17 所示。

8. 外部触发转换

如图 11-7 下方所示,规则通道、注入通道的转换可以由外部事件触发(比如定时器捕捉、EXTI 线)。如果设置了 EXTEN[1:0]或 JEXTEN[1:0]控制位不为 0,则外部事件就能够触发转换。EXTSEL[3:0]和 JEXTSEL[3:0]控制位允许应用程序选择 16 个可能的事件中的某一个,可以触发规则和注入组的采样。触发信号可以选择上升沿、下降沿或双沿触发。EXTSEL 和 JEXTSEL 的外部事件如表 11-3 所示。

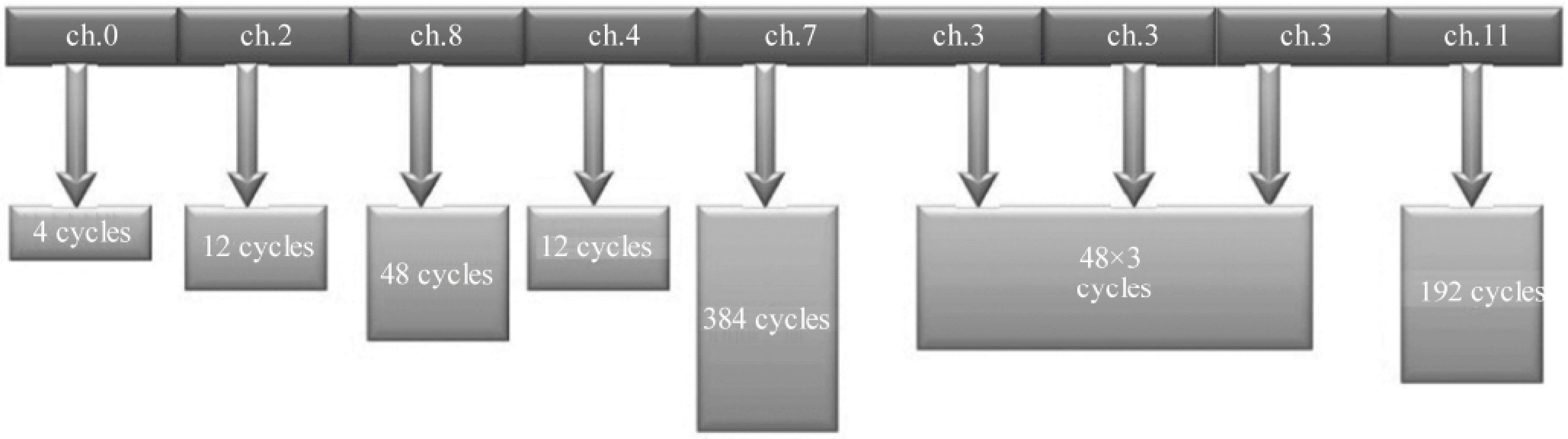


图 11-17　多通道采样时间配置案例

表 11-3　外部触发事件定义

触 发 事 件	类　　型	EXTSEL[3：0]	JEXTSEL[3：0]
TIM9_CC1	内部定时器	—	0000
TIM9_CC2	内部定时器	0000	—
TIM9_TRGO	内部定时器	0001	0001
TIM2_CC3	内部定时器	0010	—
TIM2_CC2	内部定时器	0011	—
TIM2_CC1	内部定时器	—	0011
TIM3_TRGO	内部定时器	0100	—
TIM4_CC1	内部定时器	—	0110
TIM4_CC2	内部定时器	—	0111
TIM4_CC3	内部定时器	—	1000
TIM4_CC4	内部定时器	0101	—
TIM2_TRGO	内部定时器	0110	0010
TIM3_CC1	内部定时器	0111	—
TIM3_CC3	内部定时器	1000	—
TIM3_CC4	内部定时器	—	0100
TIM4_TRGO	内部定时器	1001	0101
TIM10_CC1	内部定时器	—	1001
TIM7_TRGO	内部定时器	—	1010
TIM6_TRGO	内部定时器	1010	—
EXTI line11	外部 I/O 引脚	1111	—
EXTI line15	外部 I/O 引脚	—	1111

9. 低速转换的硬件冻结和延迟注入

ADC 的转换时钟不采用 APB 时钟，APB 时钟用于 MCU 和 ADC 之间的数据访问，当 APB 时钟太慢，不能满足 ADC 转换速率时，可以引入延迟以降低转换速率。在每个规则通道和注入组之后插入延迟，在延迟期间，ADC 转换的触发信号被忽略。注入组和规则组之间转换时不插入任何延迟，即：

- 如果在规则通道转换期间发生注入，则注入通道转换立即开始；
- 如果被注入通道打断后的规则通道要恢复执行，则立即启动，因为延时已经在上一个规则通道转换完成后添加了。

规则通道的延迟时序如图 11-18 所示，启用延迟后，在每次规则转换结束之前插入一个延迟，以便 CPU 在下一个转换完成前有时间读取 ADC_DR 中的转换数据。延迟的事件长度由 ADC_CR2 寄存器的 DELS[2：0]域指定。

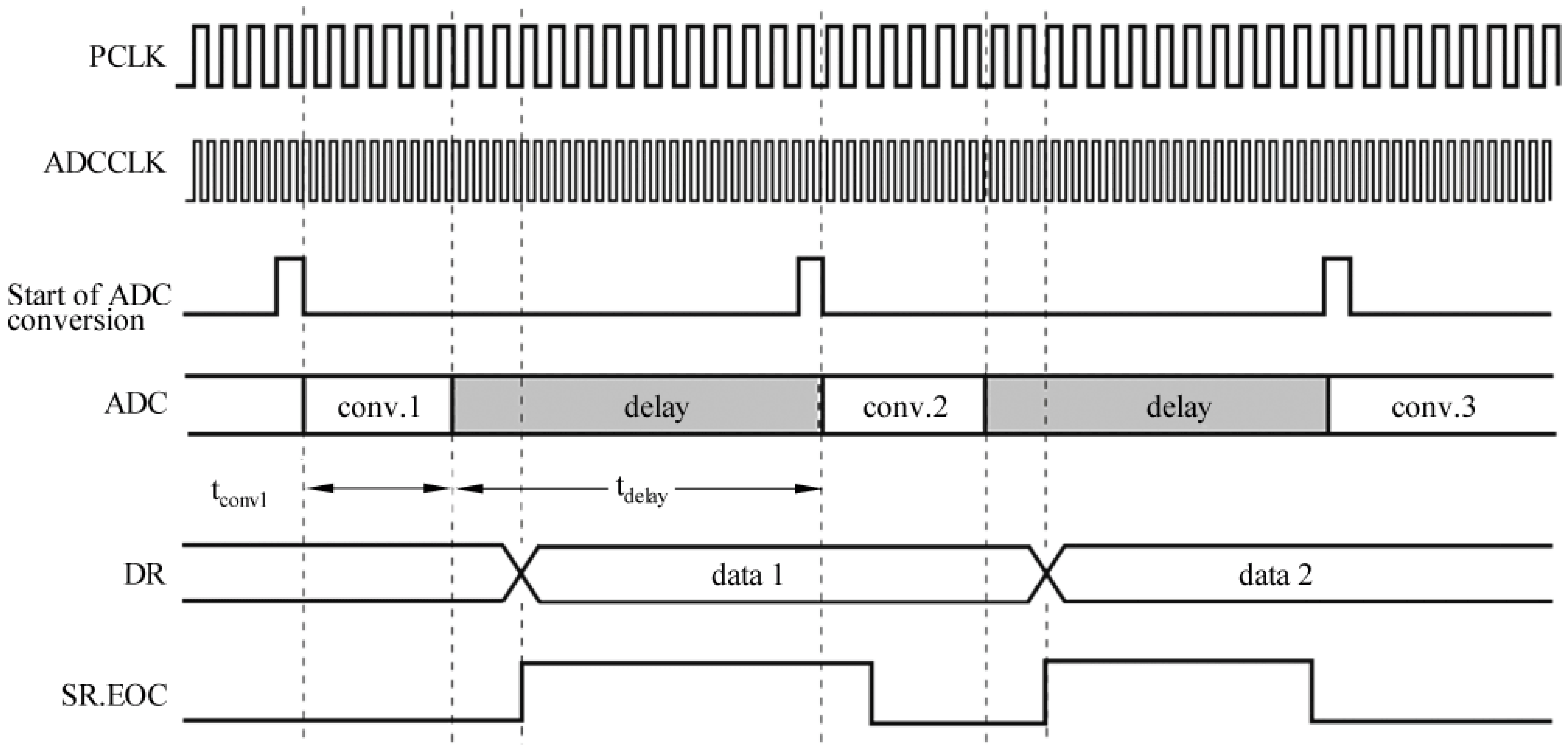


图 11-18 规则通道时延注入

自动注入转换序列后插入延迟的时序如图 11-19 所示，启用时延后，会在每个注入转换序列的末尾插入延迟。最多可以保存 5 个 ADC 转换的数据(1 个规则通道 ADC_DR 和 4 个注入通道 ADC_JDRx)，延迟的长度由 ADC_CR2 寄存器的 DELS[2：0]位配置。

配置延迟长度时有两种模式，ADC 冻结模式下，即 DELS[2：0]＝ 001 时，之前通道的所有数据处理完成后才能开始一个新的转换，即规则通道转换，读取 ADC_DR 寄存器或 EOC 后位已被清除；注入通道转换，JEOC 位被清除时。当配置成 ADC 延迟插入模式，即 DELS[2：0]＞ 001 的情况下，新转换只能在上一次转换结束，若干 APB 周期后才能开始。

10. ADC 功耗控制

STM32L152 ADC 支持上电和断电管理，以便减少 ADC 在不进行转换时的功耗，ADC 断电的时间包括：

- 注入延迟的时间(当 PDD 位置 1 时)，当注入延迟结束时，ADC 自动通电；
- ADC 等待触发事件时(PDI 位置 1 时)，ADC 在下一次触发事件到达时上电。

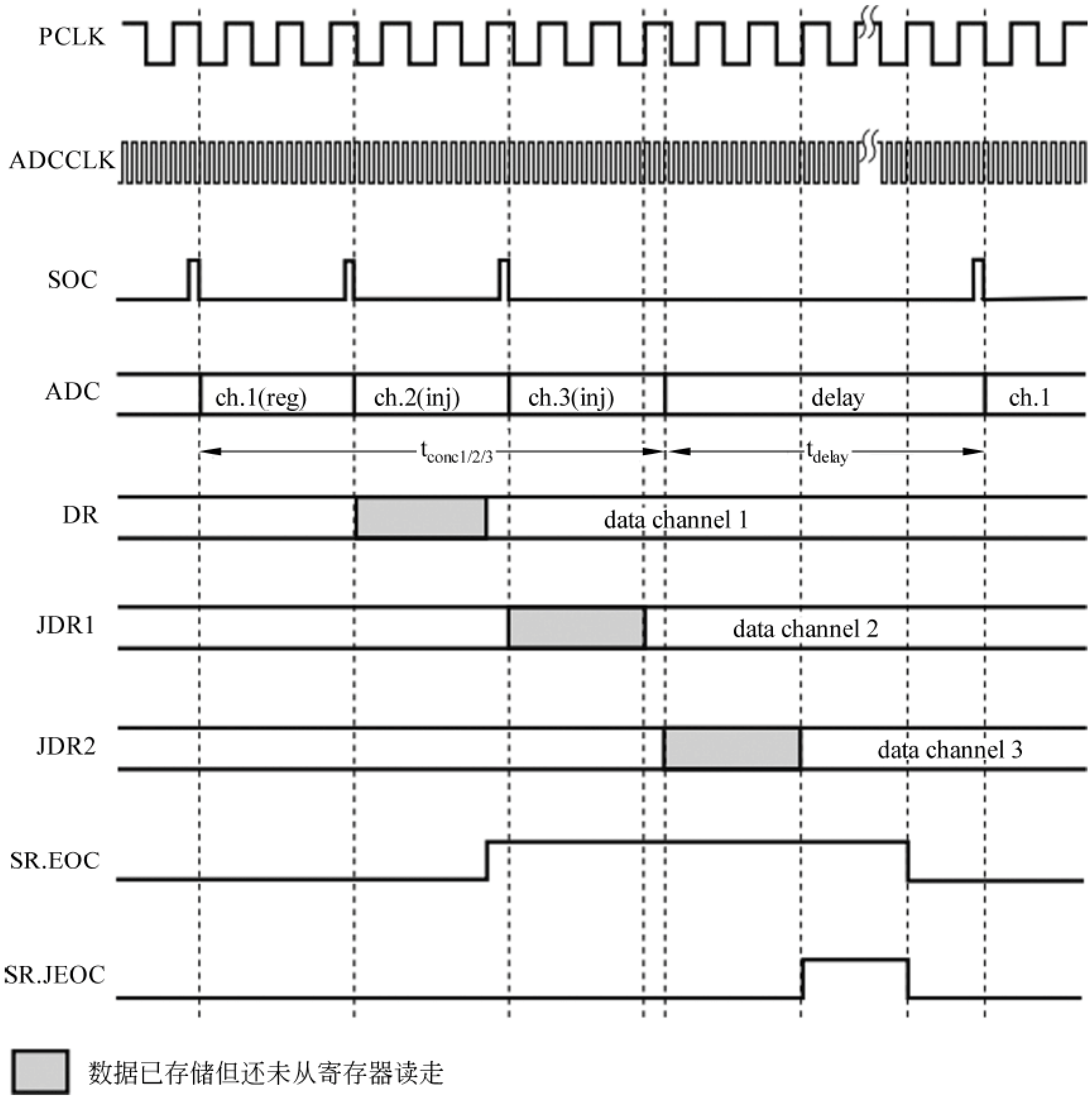


图 11-19　注入通道时延注入

在实际启动转换之前，ADC 需要一定的时间才能启动，在使用自动通断电控制之前，必须考虑到 ADC 上电的启动延迟。因此，采用扫描模式对一组通道进行转换然后断电的方式相比于单个通道转换断电效率更高。对于给定的转换序列，必须在启动之前启用 ADCCLK 时钟直到 EOC 位(或注入时的 JEOC 位)置 1 为止。

图 11-20 为不同情况下的 ADC 断电和上电时序。

11. 模拟看门狗

ADC 的模拟看门狗用于检查电压是否越界，如果被 ADC 转换的模拟电压低于低阈值或高于高阈值，AWD 模拟看门狗状态位被置 1。若 ADC_CR1 寄存器的 AWDIE 位允许产生相应中断则产生模拟看门狗中断。阈值位于 ADC_HTR 和 ADC_LTR 寄存器的最低 12 个有效位中(与对齐模式无关)。

12. 溢出错误检测

溢出检测时钟被启用，只有规则通道转换才会引起溢出错误，在一次 ADC 转换结束时，结果存储在中间缓冲区中直到它被传送到数据寄存器 ADC_DR，如果新的转换数据在

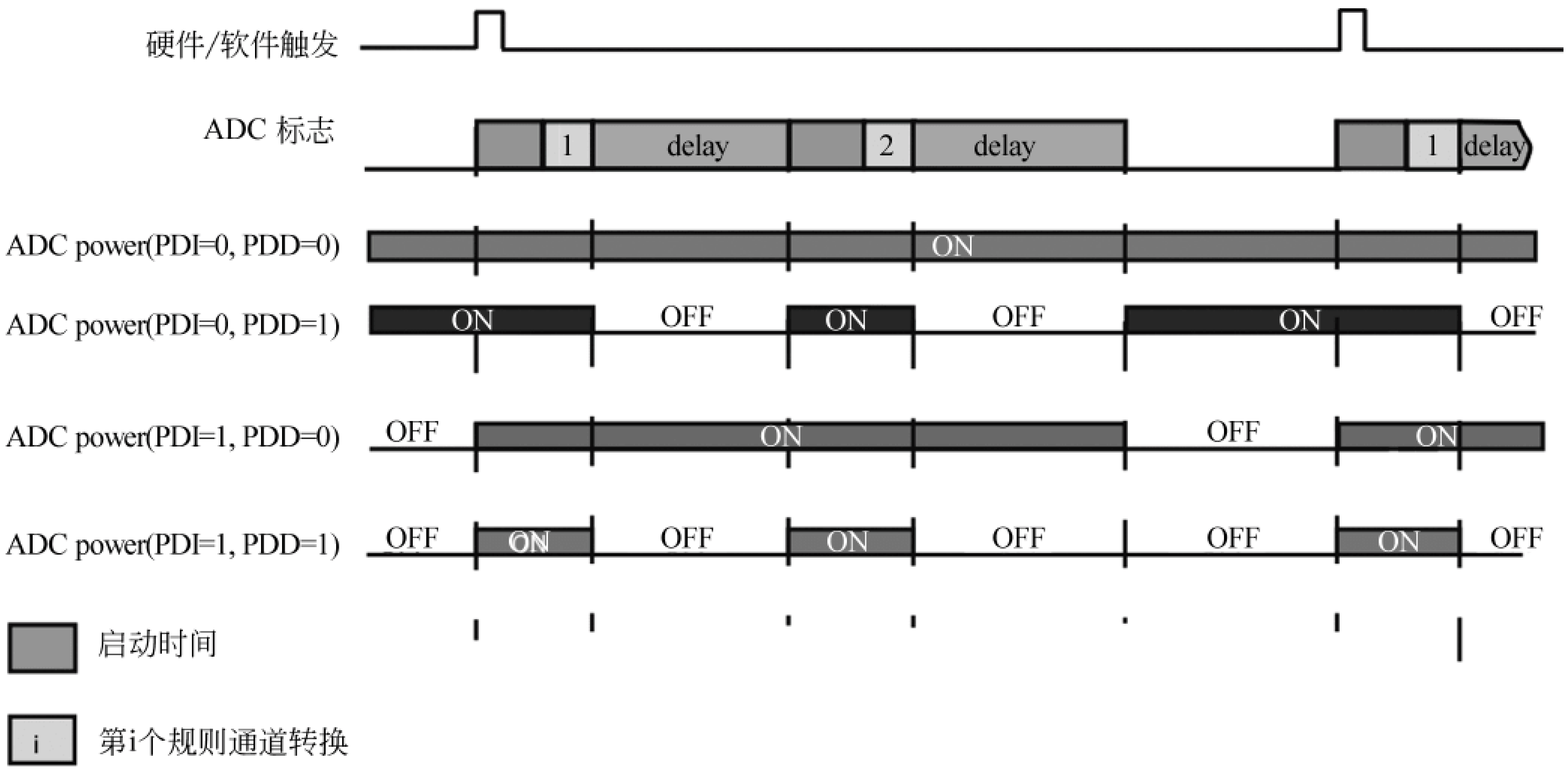


图 11-20　ADC 功耗管理模式

上一个数据传输到 ADC_DR 之前到达，则新数据会被丢弃，检测到溢出错误，ADC_SR 寄存器的 OVR 位置 1，如果 OVRIE 位置 1，则产生溢出中断。以下情况会导致溢出错误：

（1）ADCCLK 时钟和 APB 时钟不匹配，也没有正确注入时延；

（2）ADC_DR 的数据没有及时被读走，导致数据寄存器非空。

13. ADC 中断

ADC 的中断源有多个，如表 11-4 所示。规则组和注入组的转换结束时可以产生中断，当模拟看门狗状态位置 1 或溢出状态位置 1 时，均可产生中断。各种中断源可以独立灵活配置，除此之外，ADC_SR 寄存器还有 5 个状态标志用于管理 ADC，但不能产生中断。

- JCNR（注入通道未准备好）。
- RCNR（规则通道未准备好）。
- ADONS（ADON 状态）。
- JSTRT（注入组通道的转换开始）。
- STRT（规则组通道的转换开始）。

表 11-4　ADC 中断源

中　断　源	中 断 标 记	中断使能位
规则组转换结束	EOC	EOCIE
注入组转换结束	JEOC	JEOCIE
模拟看门狗	AWD	AWDIE
溢出错误	OVR	OVRIE

各种状态和中断的关系如图 11-21 所示。

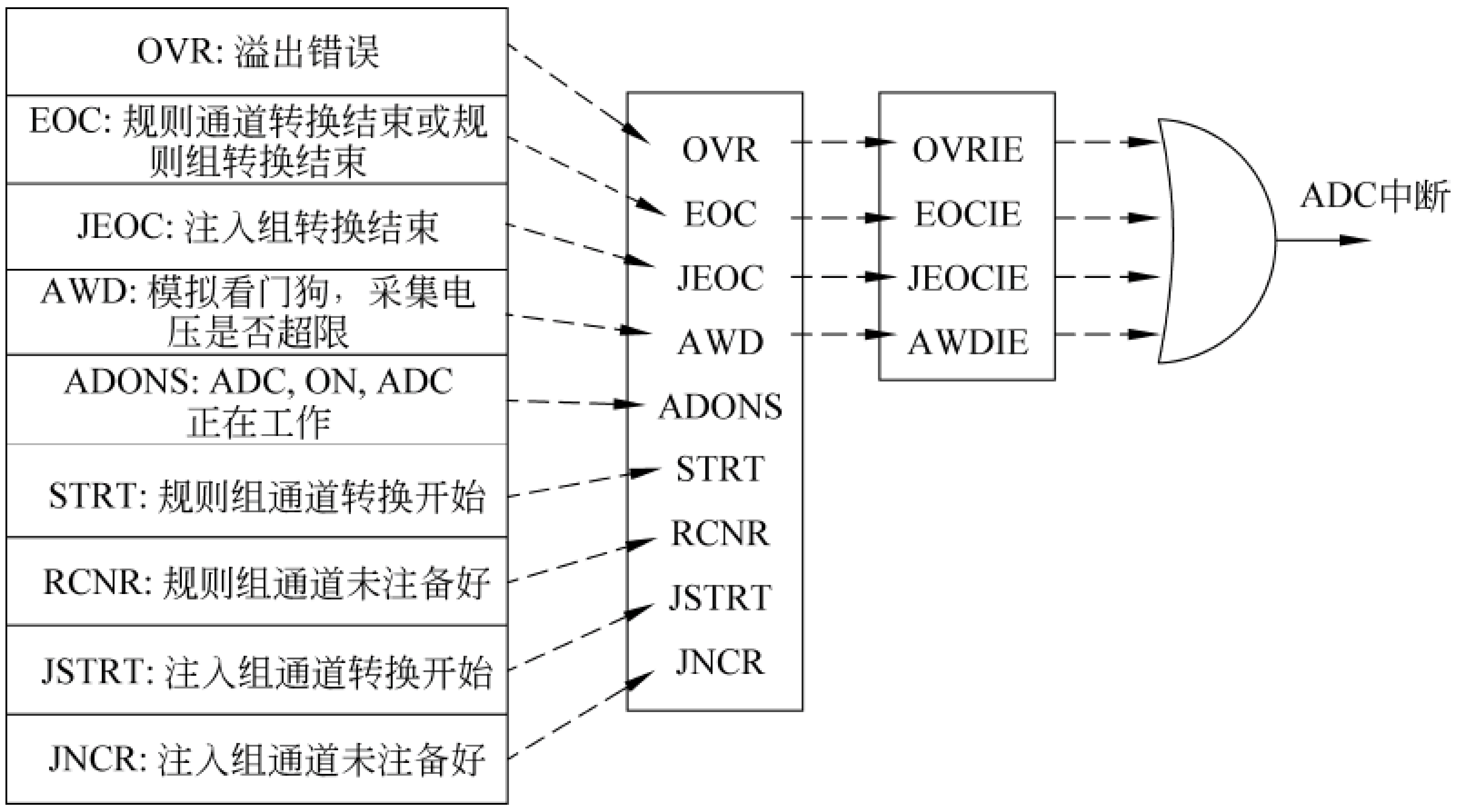


图 11-21　中断和状态标记

11.2.2　温度和电压转换

如图 11-22 所示，ADC 内部两个通道 ADC_IN16 和 ADC_IN17 分别连接到了内置温度传感器和内部参考电源上，可以通过 ADC_CCR 寄存器的 TSVREFFE 位进行使能。

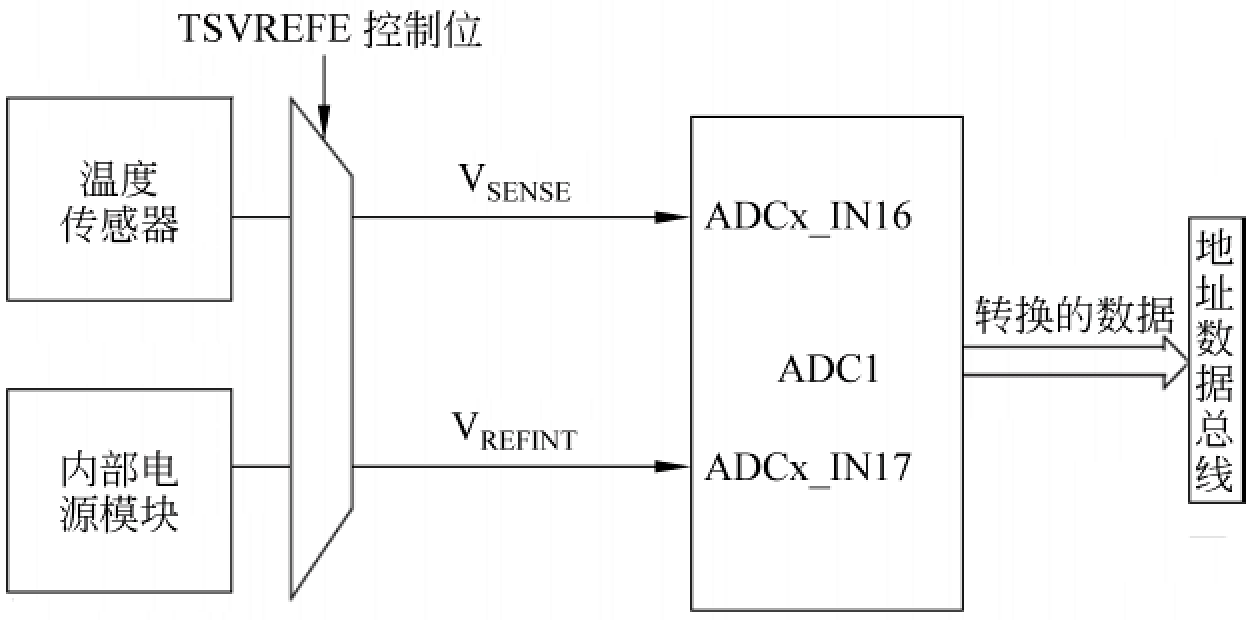


图 11-22　温度和电压测量内部连接通道

温度传感器可以用来测量 MCU 的温度(TA)。温度传感器在内部和 ADC_IN16 输入通道相连接，此通道把传感器输出的电压转换成数字值，不使用时可以将传感器置成掉电模式。温度传感器输出电压随温度线性变化而变化，温度变化曲线的偏移在不同芯片上会有不同，芯片之间温度的测量值可能会有较大差异，ST 在自定义存储区给出了每个芯片的温度的参考测量值 TS_CAL2 和 TS_CAL1，可以通过参考值补偿测量精度。

内部参考电压(VREFINT)为 ADC 和比较器提供一个稳定的参考电压，VREFINT 内部连接到 ADC_IN17 输入通道，该电压位 1.2V，同样，电压的准确值因芯片而异，可以通过读取 ST 自定义存储区的寄存器获得该电压的精确值。

1）读温度的流程

- 选择 ADC1_IN16 输入通道；
- 选择采样时间大于 4μs；
- 设置 ADC_CCR 寄存器的 TSVREFE 位，唤醒掉电模式下的温度传感器；
- 通过设置 ADON 位启动 ADC 转换；
- 读 ADC 数据寄存器 ADC_DR 的数据；
- 利用下列公式得出温度：

$$\text{Temperature}=\frac{110℃-30℃}{\text{TS_CAL2}-\text{TS_CAL1}}\times(\text{TS_DATA}-\text{TS_CAL1})+30℃$$

其中，TS_CAL2 时 110 度时的测量值，TS_CAL1 是 30 度时的测量值，可以通过读取 TS_CAL2 和 TS_CAL1 寄存器获得，TS_DATA 为通道 16 的 ADC 测量值。

2）通过内部参考电压测量系统供电 VDDA

ADC 采样中，微控制器的 VDDA 电源电压可能会发生变化（如电池供电的情况），因此 STM32 内部嵌入了一个参考电压 VREFINT，并将其连接到 ADC_IN17 上，针对每个芯片，其在 VDDA＝3V 下参考电压的 ADC 校准值保存在 ST 自定义存储区，由此我们可以通过采样 VREFINT 电压反推 VDDA 电压。实际的 VDDA 电压计算方法为：VDDA = 3 * VREFINT_CAL / VREFINT_DATA，其中，VREFINT_CAL 是 VREFINT 的校准值，VREFINT_DATA 是 ADC_IN17 的采样值。

一般 ADC 采样的范围是 0～VDDA，对于每个 ADC 通道，我们需要将 ADC 采样值转换为实际电压，其转换共识为：

$$V_{\text{CHANNELx}}=\frac{V_{\text{DDA}}}{\text{FULL_SCALE}}\times\text{ADC_DATA}_{\text{x}}$$

其中，FULL_SCALE 指的是 ADC 输出的最大值，跟分辨率有关，例如 12 比特的分辨率，FULL_SCALE ＝ $2^{12}-1=4095$。

结合上述 VDDA 的计算和校准方法，我们可以得到每个通道的实际电压的校准公式：

$$V_{\text{CHANNELx}}=\frac{3\text{V}\times\text{VREFINT_CAL}\times\text{ADC_DATA}_{\text{x}}}{\text{VREFINT_DATA}\times\text{FULL_SCALE}}$$

11.3 ADC 寄存器

ADC 涉及的寄存器如表 11-5 所示。

表 11-5 ADC 寄存器

寄存器名称	地址偏移量	作　　用	默认值
ADC_SR	0x00	ADC 状态标志	0x00000000
ADC_CR1	0x04	控制寄存器，设置扫描模式、中断允许等	0x00000000

续表

寄存器名称	地址偏移量	作　　用	默认值
ADC_CR2	0x08	控制寄存器，设置数据对齐方式、转换模式等	0x00000000
ADC_SMPR1～ADC_SMPR3	0x0c～0x14	配置 ADC 各通道的采样时间	0x00000000
ADC_JOFR1～ADC_JOFR4	0x18～0x24	配置 ADC 注入通道数据偏移量	0x00000000
ADC_HTR	0x28	模拟看门狗超限高阈值	0x00000000
ADC_LTR	0x2c	模拟看门狗超限低阈值	0x00000000
ADC_SQR1～ADC_SQR5	0x30～0x40	规则通道的数量和通道序列	0x00000000
ADC_JSQR	0x44	注入通道的数量和通道序列	0x00000000
ADC_JDR1～ADC_JDR4	0x48～0x54	存储注入通道转换数据	0x00000000
ADC_DR	0x58	存储规则通道转换数据	0x00000000
ADC_SMPR0	0x5c	ADC_IN30 和 ADC_IN31 采样时间	0x00000000
ADC_CSR	0x300	ADC_SR 的镜像	0x00000000
ADC_CCR	0x304	配置分频及内部温度、电压使能	0x00000000

1. ADC 状态寄存器 ADC_SR

ADC 状态寄存器用于标记 ADC 转换过程中的状态量，其有效域定义如图 11-23 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						JCNR	RCNR	Res.	ADONS	OVR	STRT	JSTRT	JEOC	EOC	AWD
						r	r		r	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0

图 11-23　ADC 状态寄存器

JCRN：注入通道未就绪，该位在 JSQR 寄存器被写后由硬件设置或清除，0 表示注入通道未就绪，1 表示就绪。

RCNR：规则通道未就绪，该位在 SQRx 寄存器被写后由硬件设置或清除，0 表示注入规则通道未就绪，1 表示就绪。

ADONS：ADC 开启状态，该位由硬件设置或清除，用于表示 ADC 是否准备好可以开始转换；0 表示 ADC 还未准备好，1 表示 ADC 可以开始一个转换。

OVR：溢出错误标记，该位为 1 表示有溢出错误产生，0 表示没有溢出错误；当规则通道转换数据丢失时，该位由硬件自动置 1，可以通过软件清 0。

STRT：规则通道开始位，该位由硬件在规则通道转换开始时设置，由软件清除。0 表示规则通道转换未开始；1 表示规则通道转换已开始。

JSTRT：注入通道开始位，该位由硬件在注入通道组转换开始时设置，由软件清除。0

表示注入通道组转换未开始，1 表示注入通道组转换已开始。

JEOC：注入通道转换结束位，该位由硬件在所有注入通道组转换结束时设置，由软件清除，0 表示转换未完成，1 表示转换完成。

EOC：转换结束位，该位由硬件在（规则或注入）通道组转换结束时设置，由软件清除或由读取 ADC_DR 时清除，0 表示转换未完成，1 表示转换完成。

AWD：模拟看门狗标志位，该位由硬件在转换的电压值超出了 ADC_LTR 和 ADC_HTR 寄存器定义的范围时设置，由软件清除，0 表示没有发生模拟看门狗事件，1 表示发生模拟看门狗事件。

2. ADC 控制寄存器 ADC_CR1

控制寄存器 ADC_CR1 用于设置扫描模式、中断允许等配置，其有效域定义如图 11-24 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved					OVRIE	RES[1:0]		AWDEN	JAWDEN	Reserved				PDI	PDD
					rw	rw	rw	rw	rw					rw	rw
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
DISCNUM[2:0]			JDISCEN	DISCEN	JAUTO	AWDSGL	SCAN	JEOCIE	AWDIE	EOCIE	AWDCH[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 11-24　控制寄存器 ADC_CR1

OVRIE：溢出错误中断使能，1 表示启用溢出中断，当 OVR＝1 时，产生一个 ADC 中断，0 表示禁用溢出中断。

RES[1：0]：分辨率设置，00～11 分别表示 12 位、10 位、8 位和 6 位的分辨率，这些位必须在 ADON＝1 时配置。

AWDEN：在规则通道上开启模拟看门狗，0 表示禁用模拟看门狗，1 表示使用。

JAWDEN：在注入通道上开启模拟看门狗，0 表示禁用模拟看门狗，1 表示使用。

PDI：空闲期间掉电，当 ADON＝1，该位置 1 表示在没有转换的时候 ADC 掉电，等待下一个触发事件，否则为不断电。

PDD：延迟期间掉电，当 ADON＝1 时，该位置 1 表示在两个转换之间注入的延迟期间，ADC 断电，否则为不断电。

DISCNUM[2：0]：间断模式通道计数，软件通过这些位定义在间断模式下，收到外部触发后转换规则通道的数目，编码 000～111 分别表示 1～8 个通道。

JDISCEN：开启或关闭注入通道组的间断模式，0 表示禁用间断模式，1 表示允许。

DISCEN：开启或关闭规则通道组上的间断模式，0 表示禁用间断模式，1 表示允许。

JAUTO：开启或关闭规则通道组转换结束后自动注入通道组转换，0 表示关闭，1 表示开启。

AWDSGL：开启或关闭由 AWDCH[4：0]位指定的通道上的模拟看门狗功能，0 表示在所有的通道上使用模拟看门狗，1 表示在单一通道上使用模拟看门狗。

SCAN：开启或关闭扫描模式，在扫描模式中，转换通道来自 ADC_SQRx 或 ADC_JSQRx 寄存器，0 表示关闭扫描模式，1 表示使用扫描模式。该位只能在 ADON=0 时设置。如果分别设置了 EOCIE 或 JEOCIE 位，只在最后一个通道转换完毕后才会产生 EOC 或 JEOC 中断。JEOCIE：用于禁止或允许所有注入通道转换结束后产生中断。1 表示允许 JEOC 中断，当硬件设置 JEOC 位时产生中断。

AWDIE：用于禁止或允许模拟看门狗产生中断，0 表示禁止，1 表示允许。在扫描模式下，如果看门狗检测到超范围的数值时，如果设置该位，中止 AD 转换。

EOCIE：用于禁止或允许转换结束后产生中断，0 表示禁止，1 表示允许，当硬件设置 EOC 位时产生中断。

AWDCH[4：0]：用于选择模拟看门狗保护的输入通道，00000～11010 分别表示 ADC 模拟输入通道 0～ADC 模拟输入通道 26。

3. ADC 控制寄存器 ADC_CR2

控制寄存器 ADC_CR2 用于设置数据对齐方式、连续转换位、ADC 启动位、外部触发转换等，其有效域定义如图 11-25 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	SWSTART	EXTEN		EXTSEL[3:0]				Res.	JSWSTART	JEXTEN		JEXTSEL[3:0]			
	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				ALIGN	EOCS	DDS	DMA	Res.	DELS			Res.	ADC_CFG	CONT	ADON
				rw	rw	rw	rw		rw	rw	rw		rw	rw	rw

图 11-25　控制寄存器 ADC_CR2

SWSTART：开始转换规则通道，由软件设置该位以启动转换，转换开始后硬件马上清除此位。如果在 EXTSEL[2：0]位中选择了 SWSTART 为触发事件，该位用于启动一组规则通道的转换，0 表示复位状态，1 表示开始转换规则通道。

EXTEN[1：0]：规则通道外部触发使能，由软件设置或清除，用于配置外部触发的方式，00 表示禁用外部触发，01 表示上升沿触发，10 表示下降沿触发，11 表示双沿触发，该域只能在 ADONS=1 时配置为有效。

EXTSEL[3：0]：规则通道外部触发源选择，具体配置见表 11-3。

JSWSTART：开始转换注入通道，由软件设置该位以启动转换，软件可清除此位或在转换开始后硬件马上清除此位。如果在 JEXTSEL[2：0]位中选择了 JSWSTART 为触发事件，该位用于启动一组注入通道的转换，0 表示复位状态，1 表示开始转换注入通道。

JEXTEN[1：0]：注入通道外部触发使能，由软件设置或清除，用于配置外部触发的方式，00 表示禁用外部触发，01 表示上升沿触发，10 表示下降沿触发，11 表示双沿触发，该域只能在 ADONS=1 时配置为有效。

JEXTSEL[3：0]：注入通道外部触发源选择，具体配置见表 11-3。

ALIGN：数据对齐，0 表示右对齐，1 表示左对齐。

EOCS：EOC 标记置位选择，0 表示在所有规则组通道转换完成后 EOC 置 1，1 表示规则组的每个通道转换完后 EOC 置 1。

DDS：DMA 请求选择，0 表示上一个数据转换完后不产生 DMA 请求，1 表示只要数据转换，就发起 DMA 请求。

DMA：直接存储器访问模式使能，0 表示不使用 DMA 模式，1 表示使用 DMA 模式。

DELS[2：0]：时延选择。该域用于选择所要注入的延迟的长度。000 表示不注入延迟，001 表示延迟的时间为直到转换完的数据被读走（规则通道的读数据寄存器或 EOC＝0，注入通道的 JEOC＝0），010-111 分别表示延迟时长为：7、15、21、63、127、255 个 APB 的时钟周期。在使用 DELS 配置时注意，延迟的最小值为：如果 APB 的时钟小于 ADC 时钟的 1/2，最少 15 个 APB 周期；如果 APB 时钟大于 ADC 时钟的 1/2 但小于 ADC 时钟，则最少为 7 个 APB 周期。

ADC_CFG：用于选择 ADC 的配置，使用 A 面或 B 面，0 表示使用 A 面，对应的 ADC 通道为 ADC_IN0 ～ ADC_IN31；1 表示使用通道 B 面，对应的 ADC 输入通道为 ADC_IN0b ～ ADC_IN31b。

CONT：连续转换，如果设置了此位，则转换将连续进行直到该位被清除，0 表示单次转换模式，置 1 表示连续转换模式。

ADON：A/D 转换器开关，当该位为 0 时，写入 1 将把 ADC 从断电模式下唤醒；当该位为 1 时，写入 1 将启动转换。需注意，在转换器上电至转换开始有一个延迟 tSTAB，具体值参考 STM32L152 数据手册。

4. ADC 采样时间寄存器 ADC_SMPRx（x＝1，2，3，4）

ADC_SMPRx 寄存器用于设置 ADC 各通道的采样时间，共有四个寄存器，ADC_SMPR4 寄存器的有效域定义如图 11-26 所示。每个通道使用 3 位，每个寄存器可以表示 9 个通道。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved		SMP9[2:0]			SMP8[2:0]			SMP7[2:0]			SMP6[2:0]			SMP5[2:1]	
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMP5[0]	SMP4[2:0]			SMP3[2:0]			SMP2[2:0]			SMP1[2:0]			SMP0[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 11-26　ADC_SMPR4 寄存器

SMPx[2：0]：用于独立地选择每个通道的采样时间。在采样周期中通道选择位必须保持不变。000-111 分别表示 4、9、16、24、48、96、192、384 个 ADC 时钟周期。

5. ADC 注入通道数据偏移寄存器 ADC_JOFRx（x＝1，2，3，4）

注入通道数据偏移寄存器用于设置 ADC 注入通道数据偏移量，共有 4 个注入通道，因此有 4 个寄存器，寄存器的有效域定义如图 11-27 所示。

JOFFSETx[11：0]：注入通道 x 的数据偏移，当转换注入通道时，这些位定义了用于从原始转换数据中减去的数值。转换的结果可以在 ADC_JDRx 寄存器中读出。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Reserved				JOFFSETx[11:0]											
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 11-27　注入通道数据偏移寄存器

6. ADC 看门狗高/低阈值寄存器(ADC_HTR、ADC_LRT)

看门狗阈值寄存器包括高阈值寄存器 ADC_HTR 和低阈值寄存器 ADC_LTR,用于设置 ADC 模拟看门狗的高低阈值。两个寄存器的有效域定义如图 11-28 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Reserved				HT[11:0]											
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Reserved				LT[11:0]											
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 11-28　看门狗阈值寄存器

HT[11:0]和 LT[11:0]分别用于定义模拟看门狗的阈值高限和低限。

7. ADC 规则通道序列寄存器 ADC_SQRx(x=1,2,3,4,5)

规则通道序列寄存器用于设置规则通道序列长度、对应序列中各个转换的通道编号(最多 28 个)。该寄存器有 5 个,每个通道占用 5 位用于表示一个通道编号,其中 ADC_SQR1 寄存器还有一个 L[4:0]域,其余的 4 个寄存器只有 SQx[4:0]域。ADC_SQR1 的有效域定义如图 11-29 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved							L[4:0]					SQ28[4:1]			
							rw	rw	rw	rw	rw	rw	rw	rw	rw
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
SQ28[0]	SQ27[4:0]					SQ26[4:0]					SQ25[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw				rw	rw	rw	rw	rw

图 11-29　规则通道序列寄存器 1

L[4:0]:规则通道序列的长度,用于指定总共配置了多少个通道,00000～11011 分别表示 1、2、… 28 个通道。

SQx[4:0]:规则序列中的第 x 个转换的通道编号。

8. ADC 注入序列寄存器 ADC_JSQR

注入序列寄存器用于设置注入通道序列长度、对应序列中各个转换的通道编号(最多 4 个),其有效域定义如图 11-30 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved										JL[1:0]		JSQ4[4:1]			
										rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JSQ4[0]	JSQ3[4:0]					JSQ2[4:0]					JSQ1[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 11-30　注入序列寄存器

JL[1：0]：注入通道序列长度，用于指定注入通道转换序列中的通道数目，00～11 分别表示 1～4 个通道。

JSQx[4：0]：注入序列中的第 x 个转换通道的编号。

如果 JL[1：0]的长度小于 4，则转换的序列顺序是从(4－JL)开始。例如：JL＝3，则转换序列是 JSQ1[4：0]、JSQ2[4：0]、JSQ3[4：0]、JSQ4[4：0]；如果 JL＝2，则 ADC 的转换序列是 JSQ2[4：0]、JSQ3[4：0]、JSQ4[4：0]。

9．ADC 注入数据寄存器 ADC_JDRx(x＝1，2，3，4)

注入数据寄存器用于存放 ADC 注入通道转换后的数据，最多有 4 个注入通道，因此有 4 个注入数据寄存器，其有效域定义如图 11-31 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JDATA[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

图 11-31　注入数据寄存器

JDATA[15：0]：注入转换的数据，只读，存储注入通道的转换结果。数据是左对齐或右对齐。

10．ADC 规则数据寄存器 ADC_DR

规则数据寄存器用于存放 ADC 规则通道转换后的数据，所有规则通道共享一个数据寄存器，其有效域定义如图 11-32 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

图 11-32　规则数据寄存器

DATA[15：0]：规则转换的数据，只读，包含了规则通道的转换结果。数据是左对齐或右对齐。

11．ADC 采样时间寄存器 ADC_SMPR0

采样时间寄存器 0 用于配置通道 ADC 通道 30 和通道 31 的采样时间，不是所有的 STM32L152 系列都有 30 和 31 通道。

12．ADC 通用状态寄存器 ADC_CSR

通用状态寄存器提供 ADC 状态寄存器的镜像，其有效域定义如图 11-33 所示，所有域只读且不允许清除，而是通过对 ADC_SR 寄存器的相应位写 0 来清除。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved									ADONS1	OVR1	STRT1	JSTRT1	JEOC 1	EOC1	AWD1
									r	r	r	r	r	r	r

图 11-33　通用状态寄存器

ADONS1：ADC_SR 寄存器中 ADONS 位的副本。

OVR1：ADC_SR 寄存器中 OVR 位的副本。

STRT1：ADC_SR 寄存器中 STRT 位的副本。

JSTRT1：ADC_SR 寄存器中 JSTRT 位的副本。

JEOC1：ADC_SR 寄存器中 JEOC 位的副本。

EOC1：ADC_SR 寄存器中 EOC 位的副本。

AWD1：ADC_SR 寄存器中 AWD 位的副本。

13．ADC 通用控制寄存器 ADC_CCR

该寄存器用于配置 ADC 的采样时钟预分频和内部温度、电压使能，其有效域定义如图 11-34 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								TSVREFE	Reserved					ADCPRE[1:0]	
								rw						rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved															

图 11-34　通用控制寄存器

TSVREFE：用于开启或禁止温度传感器和 VREFINT 通道。0 表示禁止，1 表示启用。

ADCPRE[1：0]：ADC 的预分频系数，00 表示 HSI 不分频，01 表示 2 分频，10 表示 4 分频，11 保留。

11.4　ADC 寄存器结构及 ADC 库函数

ADC 寄存器结构描述了固件函数库所使用的数据结构，固件库函数介绍了 ST 提供的典型库函数。

11.4.1 ADC 寄存器结构

ADC 寄存器结构，ADC_TypeDeff 和 ADC_Common_TypeDef 在文件 stm32l1xx.h 中定义如下：

```
typedef struct
{
  __IO uint32_t SR;
  __IO uint32_t CR1;
  __IO uint32_t CR2;
  __IO uint32_t SMPR1;
  __IO uint32_t SMPR2;
  __IO uint32_t SMPR3;
  __IO uint32_t JOFR1;
  __IO uint32_t JOFR2;
  __IO uint32_t JOFR3;
  __IO uint32_t JOFR4;
  __IO uint32_t HTR;
  __IO uint32_t LTR;
  __IO uint32_t SQR1;
  __IO uint32_t SQR2;
  __IO uint32_t SQR3;
  __IO uint32_t SQR4;
  __IO uint32_t SQR5;
  __IO uint32_t JSQR;
  __IO uint32_t JDR1;
  __IO uint32_t JDR2;
  __IO uint32_t JDR3;
  __IO uint32_t JDR4;
  __IO uint32_t DR;
  __IO uint32_t SMPR0;
} ADC_TypeDef;
typedef struct
{
  __IO uint32_t CSR;
  __IO uint32_t CCR;
} ADC_Common_TypeDef;
```

ADC 外设声明于文件 stm32l1xx.h：

```
#define PERIPH_BASE              ((uint32_t)0x40000000)
#define APB1PERIPH_BASE PERIPH_BASE
```

```
#define APB2PERIPH_BASE          (PERIPH_BASE + 0x10000)
#define AHBPERIPH_BASE           (PERIPH_BASE + 0x20000)
#define ADC1_BASE                (APB2PERIPH_BASE + 0x2400)
#define ADC_BASE                 (APB2PERIPH_BASE + 0x2700)
#define ADC1                     ((ADC_TypeDef * ) ADC1_BASE)
#define ADC                      ((ADC_Common_TypeDef * ) ADC_BASE)
```

ADC_InitTypeDef 和 ADC_CommonInitTypeDef 定义于文件 stm32l1xx_adc.h，用于寄存器的初始化。

```
typedef struct
{
  uint32_t ADC_Resolution;                        //分辨率
  FunctionalState ADC_ScanConvMode;               //是否扫描模式
  FunctionalState ADC_ContinuousConvMode;         //是否连续转换
  uint32_t ADC_ExternalTrigConvEdge;              //外部触发信号边沿
  uint32_t ADC_ExternalTrigConv;                  //外部触发信号
  uint32_t ADC_DataAlign;                         //对其模式
  uint8_t  ADC_NbrOfConversion;                   //转换通道数量
}ADC_InitTypeDef;
typedef struct
{
  uint32_t ADC_Prescaler;                         //分频系数
}ADC_CommonInitTypeDef;
```

ADC_Resolution 用于指定转换的分辨率，其取值为：ADC_Resolution_12b、ADC_Resolution_10b、ADC_Resolution_8b、ADC_Resolution_6b，分别表示 12 位、10 位、8 位和 6 位分辨率。

ADC_ScanConvMode 用于指定是否启用扫描模式，其取值为：ENABLE 和 DISABLE。

ADC_ContinuousConvMode 用于指定是否是连续转换，其取值为：ENABLE 和 DISABLE。

ADC_ExternalTrigConvEdge 用于指定外部触发信号的边沿，其取值为：ADC_ExternalTrigConvEdge_None、ADC_ExternalTrigConvEdge_Rising、ADC_ExternalTrigConvEdge_Falling、ADC_ExternalTrigConvEdge_RisingFalling

ADC_ExternalTrigConv 用于指定外部触发信号，规则组的触发信号定义为：

- ADC_ExternalTrigConv_T2_CC3
- ADC_ExternalTrigConv_T2_CC2
- ADC_ExternalTrigConv_T2_TRGO
- ADC_ExternalTrigConv_T3_CC1
- ADC_ExternalTrigConv_T3_CC3

- ADC_ExternalTrigConv_T3_TRGO
- ADC_ExternalTrigConv_T4_CC4
- ADC_ExternalTrigConv_T4_TRGO
- ADC_ExternalTrigConv_T6_TRGO
- ADC_ExternalTrigConv_T9_CC2
- ADC_ExternalTrigConv_T9_TRGO
- ADC_ExternalTrigConv_Ext_IT11

注入组的触发信号为：

- ADC_ExternalTrigInjecConv_T2_TRGO
- ADC_ExternalTrigInjecConv_T2_CC1
- ADC_ExternalTrigInjecConv_T3_CC4
- ADC_ExternalTrigInjecConv_T4_TRGO
- ADC_ExternalTrigInjecConv_T4_CC1
- ADC_ExternalTrigInjecConv_T4_CC2
- ADC_ExternalTrigInjecConv_T4_CC3
- ADC_ExternalTrigInjecConv_T7_TRGO
- ADC_ExternalTrigInjecConv_T9_CC1
- ADC_ExternalTrigInjecConv_T9_TRGO
- ADC_ExternalTrigInjecConv_T10_CC1
- ADC_ExternalTrigInjecConv_Ext_IT15

ADC_DataAlign 用于指定数据对齐模式，其取值为：ADC_DataAlign_Right、ADC_DataAlign_Left。

ADC_Prescaler 用于指定 ADC 采样时钟的预分频系数，其取值为：ADC_Prescaler_Div1、ADC_Prescaler_Div2、ADC_Prescaler_Div4。

11.4.2 ADC 库函数

ST 提供的 ADC 库函数如表 11-6 所示。

表 11-6 ADC 库函数

函　　数	功　　能
ADC_DeInit	初始化寄存器为复位值
ADC_Init	根据初始化模板 ADC_InitTypeDef 初始化 ADC 寄存器
ADC_StructInit	将 ADC_InitTypeDef 初始化变量设为默认值
ADC_CommonInit	根据初始化模板 ADC_CommonInitTypeDef 初始化 ADC 寄存器

续表

函　数	功　能
ADC_CommonStructInit	将 ADC_CommonInitTypeDef 初始化变量设为默认值
ADC_Cmd	启动或停止 ADC
ADC_BankSelection	ADC 输入引脚 A、B 面选择
ADC_PowerDownCmd	断电控制
ADC_DelaySelectionConfig	时延配置
ADC_AnalogWatchdogCmd	启用或停止模拟看门狗
ADC_AnalogWatchdogThresholdsConfig	模拟看门狗高低阈值配置
ADC_AnalogWatchdogSingleChannelConfig	模拟看门狗通道配置
ADC_TempSensorVrefintCmd	使能内部温度和参考电压通道
ADC_RegularChannelConfig	规则组通道配置
ADC_SoftwareStartConv	启动转换
ADC_GetSoftwareStartConvStatus	获取转换状态
ADC_EOCOnEachRegularChannelCmd	配置每次规则组通道产生 EOC
ADC_ContinuousModeCmd	连续转换模式
ADC_DiscModeChannelCountConfig	间断模式通道数量配置
ADC_DiscModeCmd	使能间断模式
ADC_GetConversionValue	获取转换结果
ADC_DMACmd	启用 DMA
ADC_DMARequestAfterLastTransferCmd	配置 DMA 的产生时机
ADC_InjectedChannelConfig	注入通道配置
ADC_InjectedSequencerLengthConfig	注入通道序列长度设置
DC_SetInjectedOffset	注入通道偏移量
ADC_ExternalTrigInjectedConvConfig	注入通道外部触发配置
ADC_ExternalTrigInjectedConvEdgeConfig	诸如通道外部触发信号边沿设置
ADC_SoftwareStartInjectedConv	启动注入通道转换
ADC_GetSoftwareStartInjectedConvCmd	获取注入通道转换状态
ADC_AutoInjectedConvCmd	启用自动注入
ADC_InjectedDiscModeCmd	使能注入通道间断模式
ADC_GetInjectedConversionValue	获取注入通道转换值
ADC_ITConfig	配置 ADC 中断源

续表

函　　数	功　　能
ADC_GetFlagStatus	获取状态标志结果
ADC_ClearFlag	清除状态位
ADC_GetITStatus	获取中断源
ADC_ClearITPendingBit	清除中断源

1）函数 ADC_DeInit

函数功能：初始化 ADC1 寄存器为复位值。

函数原型：void ADC_DeInit(ADC_TypeDef * ADCx)。

输入参数 ADCx：需要初始化的 ADC 外设，取值为 ADC1。

2）函数 ADC_Init

函数功能：根据 ADC_InitStruct 指定的参数初始化 ADC1 寄存器。

函数原型：void ADC_Init(ADC_TypeDef * ADCx, ADC_InitTypeDef * ADC_InitStruct)。

输入参数 ADCx：需要初始化的 ADC 外设，取值为 ADC1。

输入参数 ADC_InitStruct：指向 ADC_InitTypeDef 结构体的指针。

3）函数 ADC_StructInit

函数功能：使用默认值初始化 ADC_InitStruct 成员。

函数原型：void ADC_StructInit(ADC_InitTypeDef * ADC_InitStruct)。

输入参数 ADC_InitStruct：指向 ADC_InitTypeDef 结构的指针。

默认值为：

```
ADC_Resolution = ADC_Resolution_12b;
ADC_ScanConvMode = DISABLE;
ADC_ContinuousConvMode = DISABLE;
ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_None;
ADC_ExternalTrigConv = ADC_ExternalTrigConv_T2_CC2;
ADC_DataAlign = ADC_DataAlign_Right;
ADC_NbrOfConversion = 1;
```

4）函数 ADC_CommonInit

函数功能：根据 ADC_CommonInitStruct 指定的参数初始化 ADC 外设。

函数原型：void ADC_CommonInit(ADC_CommonInitTypeDef * ADC_CommonInitStruct)。

输入参数 ADC_CommonInitStruct：指向 ADC_CommonInitTypeDef 结构的指针。

5）函数 ADC_CommonStructInit

函数功能：使用默认值初始化 ADC_CommonInitStruct 成员。

函数原型：void ADC_CommonStructInit(ADC_CommonInitTypeDef * ADC_CommonInitStruct)。

输入参数 ADC_CommonInitStruct：指向 ADC_CommonInitTypeDef 结构的指针。

默认值 ADC_Prescaler = ADC_Prescaler_Div1。

6）函数 ADC_Cmd

函数功能：启用或停止 ADC。

函数原型：void ADC_Cmd(ADC_TypeDef * ADCx, FunctionalState NewState)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 NewState：ADC 的状态，取值为 ENABLE 或 DISABLE。

7）函数 ADC_BankSelection

函数功能：选择 ADC 输入通道来源。

函数原型：void ADC_BankSelection(ADC_TypeDef * ADCx, uint8_t ADC_Bank)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 ADC_Bank：要选择的 ADC 输入通道，取值为 ADC_Bank_A 和 ADC_Bank_B，当选用 ADC_Bank_A 时，ADC 通道为 ADC_IN0～ADC_IN31，当选用 ADC_Bank_B 时，ADC 通道为 ADC_IN0b～ADC_IN31b。

8）函数 ADC_PowerDownCmd

函数功能：在注入延迟或空闲期间启用或禁用 ADC 掉电功能。

函数原型：void ADC_PowerDownCmd(ADC_TypeDef * ADCx, uint32_t ADC_PowerDown, FunctionalState NewState)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 ADC_PowerDown：ADC 掉电配置，取值范围为：

- ADC_PowerDown_Delay：ADC 在延迟阶段断电。
- ADC_PowerDown_Idle：ADC 在空闲阶段关闭。
- ADC_PowerDown_Idle_Delay：ADC 在延迟和空闲阶段断电。

输入参数 NewState：ADCx 掉电的新状态，取值为 ENABLE 或 DISABLE。

9）函数 ADC_DelaySelectionConfig

函数功能：配置 ADC 注入延迟的大小。

函数原型：void ADC_DelaySelectionConfig(ADC_TypeDef * ADCx, uint8_t ADC_DelayLength)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 ADC_DelayLength：注入的延迟长度，取值为：

- ADC_DelayLength_None：没有延迟；
- ADC_DelayLength_Freeze：延迟到读取转换后的数据；
- ADC_DelayLength_7Cycles：延迟长度等于 7 个 APB 时钟周期；
- ADC_DelayLength_15Cycles：延迟长度等于 15 个 APB 时钟周期；
- ADC_DelayLength_31Cycles：延迟长度等于 31 个 APB 时钟周期；
- ADC_DelayLength_63Cycles：延迟长度等于 63 个 APB 时钟周期；
- ADC_DelayLength_127Cycles：延迟长度等于 127 个 APB 时钟周期；

• ADC_DelayLength_255Cycles：延迟长度等于 255 个 APB 时钟周期。

10）函数 ADC_TempSensorVrefintCmd

函数功能：启用/禁用 ADC 与温度传感器和 Vrefint 源之间的内部连接。

函数原型：void ADC_TempSensorVrefintCmd(FunctionalState NewState)。

输入参数 NewState：温度传感器和 Vrefint 是否连接，取值为 ENABLE 或 DISABLE。

11）函数 ADC_RegularChannelConfig

函数功能：为所选 ADC 规则通道配置其对应的规则组序号及其采样时间。

函数原型：void ADC_RegularChannelConfig(ADC_TypeDef * ADCx, uint8_t ADC_Channel, uint8_t Rank, uint8_t ADC_SampleTime)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 ADC_Channel：要配置的 ADC 通道，取值为 ADC_Channel_x(x=0～27)或 ADC_Channel_xb(x=0～12)。

输入参数 Rank：所要配置的通道在规则组通道的次序，取值范围为 1～28。

输入参数 ADC_SampleTime：所选择通道的采样时间，取值范围为：

• ADC_SampleTime_4Cycles：采样时间等于 4 个周期；
• ADC_SampleTime_9Cycles：采样时间等于 9 个周期；
• ADC_SampleTime_16Cycles：采样时间等于 16 个周期；
• ADC_SampleTime_24Cycles：采样时间等于 24 个周期；
• ADC_SampleTime_48Cycles：采样时间等于 48 个周期；
• ADC_SampleTime_96Cycles：采样时间等于 96 个周期；
• ADC_SampleTime_192Cycles：采样时间等于 192 个周期；
• ADC_SampleTime_384Cycles：采样时间等于 384 个周期。

12）函数 ADC_SoftwareStartConv

函数功能：软件配置启动规则通道的转换。

函数原型：void ADC_SoftwareStartConv(ADC_TypeDef * ADCx)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

13）函数 ADC_GetSoftwareStartConvStatus

函数功能：获取所选的 ADC 规则转换软件启动的状态值。

函数原型：FlagStatus ADC_GetSoftwareStartConvStatus(ADC_TypeDef * ADCx)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

返回值：ADC 开始转换或没有转换，取值为 SET 或 RESET。

14）函数 ADC_EOCOnEachRegularChannelCmd

函数功能：启用或禁用每次规则通道转换完成产生 EOC。

函数原型：void ADC_EOCOnEachRegularChannelCmd(ADC_TypeDef * ADCx, FunctionalState NewState)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 NewState：所选 ADC EOC 标志是否在每次转换完后产生，取值为 ENABLE

或 DISABLE。

15）函数 ADC_ContinuousModeCmd

函数功能：启用或禁用 ADC 连续转换模式。

函数原型：void ADC_ContinuousModeCmd(ADC_TypeDef * ADCx，FunctionalState NewState)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 NewState：ADC 连续转换功能的状态，取值为 ENABLE 或 DISABLE。

16）函数 ADC_DiscModeChannelCountConfig

函数功能：配置所选 ADC 规则通道组的间断模式。

函数原型：void ADC_DiscModeChannelCountConfig(ADC_TypeDef * ADCx，uint8_t Number)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 Number：规则通道间断模式的转换通道数量，取值范围为 1～8。

17）函数 ADC_DiscModeCmd

函数功能：启用或禁用规则通道组的间断模式。

函数原型：void ADC_DiscModeCmd(ADC_TypeDef * ADCx，FunctionalState NewState)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 NewState：间断模式的启用状态，取值为 ENABLE 或 DISABLE。

18）函数 ADC_GetConversionValue

函数功能：获取规则通道的 ADC 转换结果数据。

函数原型：uint16_t ADC_GetConversionValue(ADC_TypeDef * ADCx)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

返回值：ADC 数据转换结果。

19）函数 ADC_InjectedChannelConfig

函数功能：为所选 ADC 规则通道配置其对应的规则组序号及其采样时间。

函数原型：void ADC_InjectedChannelConfig(ADC_TypeDef * ADCx，uint8_t ADC_Channel，uint8_t Rank，uint8_t ADC_SampleTime)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 ADC_Channel：要配置的 ADC 通道，取值为 ADC_Channel_x(x=0～27)或 ADC_Channel_xb(x=0～12)。

输入参数 Rank：所要配置的通道在规则组通道的次序，取值范围为 1～4。

输入参数 ADC_SampleTime：所选择通道的采样时间，取值范围与 ADC_RegularChannelConfig 函数的 ADC_SampleTime 相同。

20）函数 ADC_InjectedSequencerLengthConfig

函数功能：用于配置注入通道组的通道数量。

函数原型：void ADC_InjectedSequencerLengthConfig(ADC_TypeDef * ADCx，uint8

_t Length)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 Length：注入组通道数量，取值范围为 1～4。

21）函数 ADC_SetInjectedOffset

函数功能：设置注入通道转换值的偏移量。

函数原型：void ADC_SetInjectedOffset(ADC_TypeDef * ADCx，uint8_t ADC_InjectedChannel，uint16_t Offset)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 ADC_InjectedChannel：ADC 注入通道编号，取值范围为：

- ADC_InjectedChannel_1：已选择注入的 Channel1。
- ADC_InjectedChannel_2：已选择注入的 Channel2。
- ADC_InjectedChannel_3：已选择注入的 Channel3。
- ADC_InjectedChannel_4：已选择注入的 Channel4。

输入参数 Offset：所选 ADC 注入通道的偏移值，12 位。

22）函数 ADC_ExternalTrigInjectedConvConfig

函数功能：为注入通道转换配置 ADCx 外部触发源。

函数原型：void ADC_ExternalTrigInjectedConvConfig(ADC_TypeDef * ADCx，uint32_t ADC_ExternalTrigInjecConv)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 ADC_ExternalTrigInjecConv：指定 ADC 转换的触发源，其取值与 ADC 初始化模板中 ADC_ExternalTrigConv 的取值相同。

23）函数 ADC_ExternalTrigInjectedConvEdgeConfig

函数功能：注入通道触发的外部触发沿配置。

函数原型：void ADC_ExternalTrigInjectedConvEdgeConfig(ADC_TypeDef * ADCx，uint32_t ADC_ExternalTrigInjecConvEdge)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 ADC_ExternalTrigInjecConvEdge：指定 ADC 转换的触发源，其取值与 ADC 初始化模板中 ADC_ExternalTrigInjecConvEdge 的取值相同。

24）函数 ADC_SoftwareStartInjectedConv

函数功能：注入通道软件启动转换设置。

函数原型：void ADC_SoftwareStartInjectedConv(ADC_TypeDef * ADCx)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

25）函数 ADC_GetSoftwareStartInjectedConvCmdStatus

函数功能：获取注入通道软件启动转换的状态。

函数原型：FlagStatus ADC_GetSoftwareStartInjectedConvCmdStatus(ADC_TypeDef * ADCx)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

返回值为 ADC 是否已启动注入通道转换，取值范围为 SET 或 RESET。

26）函数 ADC_AutoInjectedConvCmd

函数功能：使能或禁用 ADC 在规则通道转换完后自动执行注入组转换功能。

函数原型：void ADC_AutoInjectedConvCmd(ADC_TypeDef * ADCx，FunctionalState NewState)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 NewState：自动注入通道转换的状态，取值为 ENABLE 或 DISABLE。

27）函数 ADC_InjectedDiscModeCmd

函数功能：为指定的 ADC 外设启用或禁用注入组通道的间断模式。

函数原型：void ADC_InjectedDiscModeCmd(ADC_TypeDef * ADCx，FunctionalState NewState)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 NewState：注入通道间断模式的状态，取值为 ENABLE 或 DISABLE。

28）函数 ADC_GetInjectedConversionValue

函数功能：获取指定 ADC 外设的注入通到转换结果。

函数原型：uint16_t ADC_GetInjectedConversionValue(ADC_TypeDef * ADCx，uint8_t ADC_InjectedChannel)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 ADC_InjectedChannel：注入通道的编号，取值为：

- ADC_InjectedChannel_1：注入通道 Channel1；
- ADC_InjectedChannel_2：注入通道 Channel2；
- ADC_InjectedChannel_3：注入通道 Channel3；
- ADC_InjectedChannel_4：注入通道 Channel4。

29）函数 ADC_ITConfig

函数功能：启用或禁用指定的 ADC 中断。

函数原型：void ADC_ITConfig(ADC_TypeDef * ADCx，uint16_t ADC_IT，FunctionalState NewState)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 ADC_IT：指定要启用或禁用的 ADC 中断源，取值为：

- ADC_IT_EOC：转换结束中断。
- ADC_IT_AWD：模拟看门狗中断。
- ADC_IT_JEOC：注入转换结束中断。
- ADC_IT_OVR：溢出中断。

输入参数 NewState：指定 ADC 中断的状态，取值为 ENABLE 或 DISABLE。

30）函数 ADC_GetFlagStatus

函数功能：检查指定的 ADC 标志是否设置。

函数原型：FlagStatus ADC_GetFlagStatus(ADC_TypeDef * ADCx，uint16_t ADC_

FLAG)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 ADC_FLAG：指定要检查的标志，取值为：

- ADC_FLAG_AWD：模拟看门狗标志；
- ADC_FLAG_EOC：转换结束标志；
- ADC_FLAG_JEOC：注入组转换结束标志；
- ADC_FLAG_JSTRT：注入组转换开始标志；
- ADC_FLAG_STRT：规则组开始转换标志；
- ADC_FLAG_OVR：溢出标志；
- ADC_FLAG_ADONS：ADC ON 状态；
- ADC_FLAG_RCNR：规则通道转换未就绪；
- ADC_FLAG_JCNR：注入通道转换未就绪。

返回值：ADC_FLAG 的状态，取值为 SET 或 RESET。

31）函数 ADC_ClearFlag

函数功能：清除 ADC 标志。

函数原型：void ADC_ClearFlag(ADC_TypeDef * ADCx, uint16_t ADC_FLAG)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 ADC_FLAG：指定要清除的标志，取值为：ADC_FLAG_AWD、ADC_FLAG_EOC、ADC_FLAG_JEOC、ADC_FLAG_JSTRT、ADC_FLAG_STRT 和 ADC_FLAG_OVR。

32）函数 ADC_GetITStatus

函数功能：检查指定的 ADC 中断是否发生。

函数原型：ITStatus ADC_GetITStatus(ADC_TypeDef * ADCx, uint16_t ADC_IT)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 ADC_IT：指定要检查的中断源，取值为：

- ADC_IT_EOC：规则通道转换结束中断；
- ADC_IT_AWD：模拟看门狗中断；
- ADC_IT_JEOC：注入通道转换结束中断；
- ADC_IT_OVR：溢出中断。

返回值：ADC_IT 的状态，取值为 SET 或 RESET。

33）函数 ADC_ClearITPendingBit

函数功能：清除指定的 ADC 中断挂起位。

函数原型：void ADC_ClearITPendingBit(ADC_TypeDef * ADCx, uint16_t ADC_IT)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 ADC_IT：指定要清除的中断源，取值与函数 ADC_GetITStatus 的参数 ADC_IT 相同。

11.5 ADC 案例

11.5.1 ADC 寄存器操作案例

【例 11-1】 ADC_Init 函数的寄存器操作实现。

```
void ADC_Init(ADC_TypeDef * ADCx, ADC_InitTypeDef * ADC_InitStruct)
{
  uint32_t tmpreg1 = 0;
  uint8_t tmpreg2 = 0;
  //ADCx CR1 寄存器配置
  tmpreg1 = ADCx->CR1;                                        //获取 ADCx CR1 寄存器值
  tmpreg1 &= CR1_CLEAR_MASK;                                  //清除 RES 和 SCAN 域
  //根据 ADC_ScanConvMode 配置域,根据 ADC_Resolution 配置 RES 域
  tmpreg1 |= (uint32_t)(((uint32_t)ADC_InitStruct->ADC_ScanConvMode << 8) |
            ADC_InitStruct->ADC_Resolution);
  ADCx->CR1 = tmpreg1;                                        //写回到寄存器 ADCx CR1
  //配置 ADCx CR2 寄存器
  tmpreg1 = ADCx->CR2;                                        //获取 ADCx CR2 寄存器值
  //清除 CONT, ALIGN, EXTEN and EXTSEL 域
  tmpreg1 &= CR2_CLEAR_MASK;
  //根据 ADC_DataAlign 配置 ALIGN 域,根据 ADC_ExternalTrigConvEdge 配置 EXTEB 域,
  //根据 ADC_ExternalTrigConv 配置 EXTSEL 域,根据 ADC_ContinuousConvMode 配置 CONT 域
  tmpreg1 |= (uint32_t)(ADC_InitStruct->ADC_DataAlign
          | ADC_InitStruct->ADC_ExternalTrigConv
          | ADC_InitStruct->ADC_ExternalTrigConvEdge
          | ((uint32_t)ADC_InitStruct->ADC_ContinuousConvMode << 1));
  ADCx->CR2 = tmpreg1;                                        //写回寄存器 ADCx CR2
  //配置 ADCx SQR1 寄存器
  tmpreg1 = ADCx->SQR1;                                       //获取寄存器 ADCx SQR1 的值
  tmpreg1 &= SQR1_L_RESET;                                    //清除 L 域
  //根据 ADC_NbrOfConversion 配置规则组通道数量 L
  tmpreg2 |= (uint8_t)(ADC_InitStruct->ADC_NbrOfConversion - (uint8_t)1);
  tmpreg1 |= ((uint32_t)tmpreg2 << 20);
  ADCx->SQR1 = tmpreg1;                                       //写回到寄存器 ADCx SQR1
}
```

【例 11-2】 ADC_Cmd 寄存器操作实现。

```
void ADC_Cmd(ADC_TypeDef * ADCx, FunctionalState NewState)
{
```

```
    if (NewState != DISABLE)
      ADCx->CR2 |= (uint32_t)ADC_CR2_ADON;                    //设置 ADON 域给 ADC 上电
    else
      ADCx->CR2 &= (uint32_t)(~ADC_CR2_ADON);                 //关闭 ADC
  }
```

11.5.2 ADC 库函数操作案例

1）ADC 配置流程：

- 配置 ADC 的 GPIO 为模拟输入；
- 使能 HSI 时钟，要等待 HSI 时钟开启；
- 使能 ADC 时钟；
- 配置 ADC 相关参数（转换精度，转换模式，字节对齐）；
- 配置 ADC 通道和采样时钟；
- 配置 ADC 采样频率（预分频参数）；
- 配置 ADC 中断向量相关参数；
- 开启 ADC 的 EOC 中断；
- 给 ADC 上电，并检测 ADC 是否准备好；
- 软件开启 ADC。

2）ADC 中断处理：

- 判断 EOC 中断标志位；
- 对 EOC 中断清零；
- 对转换数值处理；
- ADC 上电；
- 检测 ADONS 标志位，等待 ADC 准备好；
- 开启软件打开方式转换。

【例 11-3】 ADC 初始化配置。

```
void ADC_InitConfiguration(void)
{
  ADC_InitTypeDef ADC_InitStructure;
  RCC_HSICmd(ENABLE);
  while (RCC_GetFlagStatus(RCC_FLAG_HSIRDY) == RESET);
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
  RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);
  GPIO_InitStructure.GPIO_Speed=GPIO_Speed_40MHz;
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN;
  GPIO_Init(GPIOA, &GPIO_InitStructure);
```

```
    ADC_DeInit(ADC1);
    ADC_StructInit(&ADC_InitStructure);
    ADC_InitStructure.ADC_Resolution = ADC_Resolution_12b;
    ADC_InitStructure.ADC_ScanConvMode = DISABLE;
    ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
    ADC_InitStructure.ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_None;
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
    ADC_InitStructure.ADC_NbrOfConversion = 1;
    ADC_Init(ADC1, &ADC_InitStructure);
    ADC_DelaySelectionConfig(ADC1, ADC_DelayLength_Freeze);
    ADC_PowerDownCmd(ADC1, ADC_PowerDown_Idle_Delay, ENABLE);
    ADC_RegularChannelConfig(ADC1, ADC_Channel_5, 1, ADC_SampleTime_96Cycles );

    //ADC_ITConfig(ADC1,ADC_IT_EOC,ENABLE);                    //中断使能
    ADC_Cmd(ADC1, ENABLE);
    while (ADC_GetFlagStatus(ADC1, ADC_FLAG_ADONS) == RESET);
}
```

启用 ADC 转换，读取数据的代码如下：

```
ADC_SoftwareStartConv(ADC1);
while (ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == RESET) ;
uint16_t  ADCdata = ADC_GetConversionValue(ADC1);
```

Chapter 12

第 12 章　低功耗技术

【导读】 低功耗时 STM32L 系列处理器的优势，在物联网电池供电系统中对于处理器的功耗要求较高。本章首先介绍了处理器的动态功耗和静态功耗，然后针对 STM32L1xx 系列处理器的电压管理、时钟管理、低功耗模式进行了介绍，并对 PWR 控制器的寄存器、库函数及典型使用方法进行了介绍。

12.1　处理器功耗的构成/类型

电源对电子设备的重要性不言而喻，它是保证系统稳定运行的基础，在很多应用场合中都对电子设备的功耗要求非常苛刻，如某些传感器信息采集设备，仅靠小型的电池提供电源，要求工作长达数年之久，且期间不需要任何维护。由于智慧穿戴设备的小型化要求，电池体积不能太大导致容量也比较小，所以也很有必要从控制功耗入手，提高设备的续行时间。STM32L 系列针对低功耗处理进行了优化，有专门的电源管理和低功耗运行模式。

功耗的构成主要有动态功耗、静态功耗、浪涌功耗这三种。

12.1.1　动态功耗

动态功耗主要是外围电路的器件功耗，包括：开关功耗（翻转功耗）和短路功耗（内部功耗）。

1. 开关功耗

开关功耗指的是数字 CMOS 电路中，对负载电容进行充放电时消耗的功耗，比如对于图 12-1 的 CMOS 非门中，当 $V_{in}=0$ 时，上面的 PMOS 导通，下面的 NMOS 截止；V_{DD} 对负载电容 C_{load} 进行充电，充电完成后，V_{out} 的电平为高电平。

当 $V_{in}=1$ 时，上面的 PMOS 截止，下面的 NMOS 导通，负载电容通过 NMOS 进行放电，放电完成后，V_{out} 的电平为低电平。充放电形成了开关功耗，开关功耗 P_{switch} 的计算公式为：

$$P_{switch}=\frac{1}{2}V_{DD}^{2}C_{load}T_{r}$$

其中，V_{DD}为供电电压，C_{load}为后级电路等效的电容负载大小，T_r 为输入信号的翻转频率。

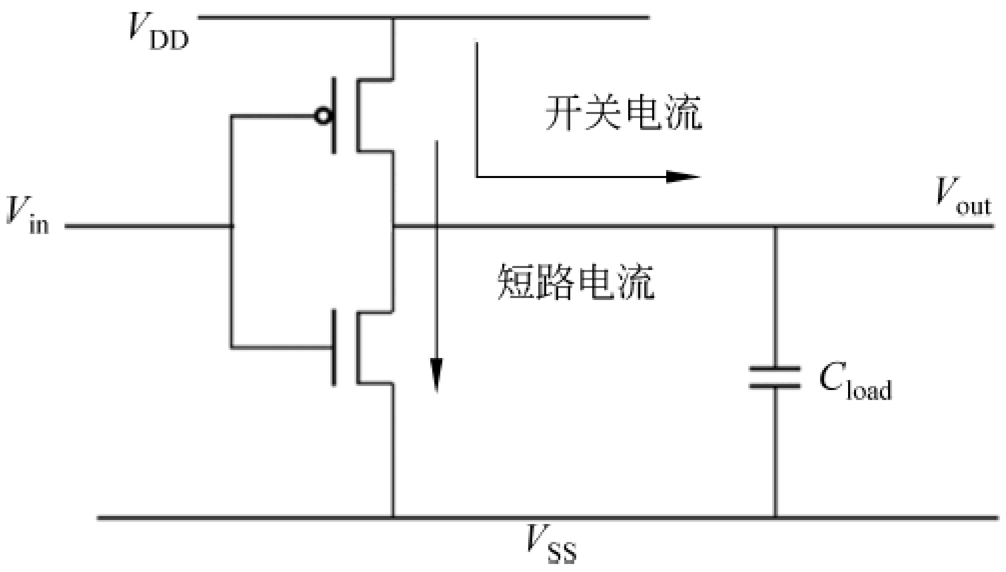


图 12-1　CMOS 非门

2. 短路功耗

短路功耗也称为内部功耗，在输入信号进行翻转时，信号的翻转不可能瞬时完成，因此 PMOS 和 NMOS 不可能总是一个截止另外一个导通，会存在一段时间，PMOS 和 NMOS 同时导通，从而导致电源 VDD 到地 VSS 之间就有短路电流，如图 12-2 的反向器电路所示。

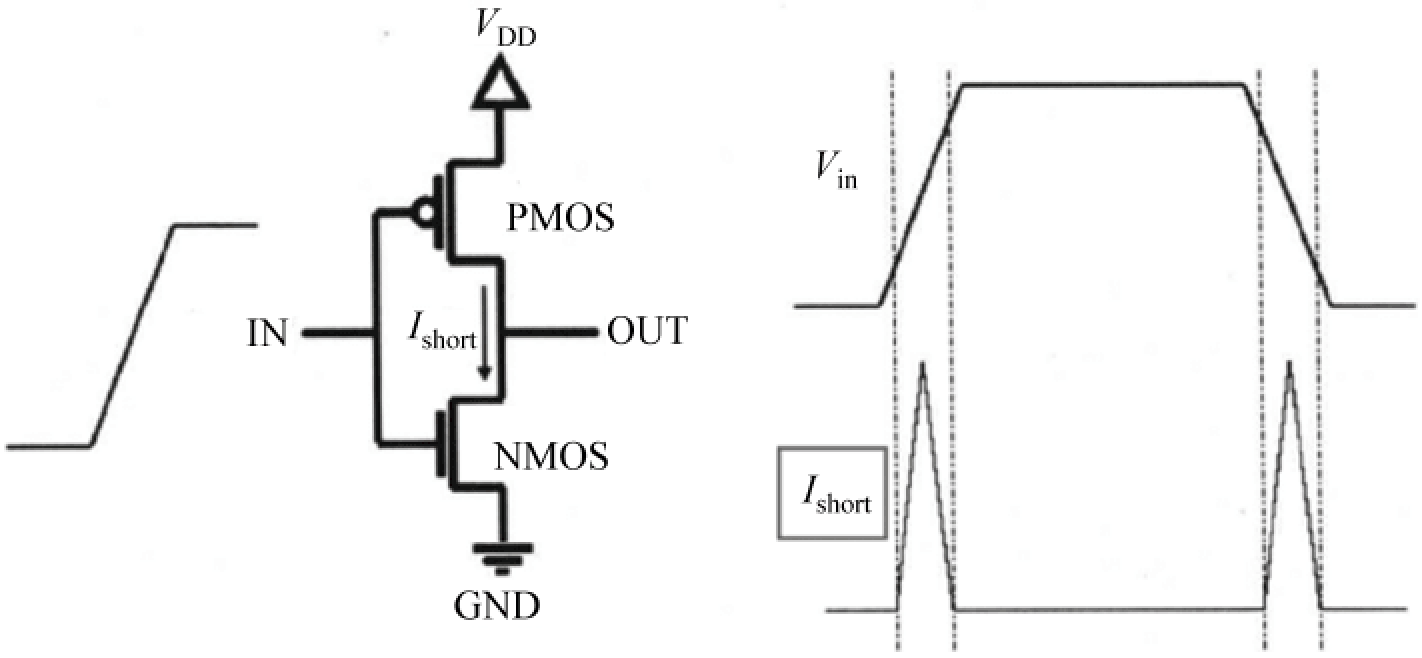


图 12-2　反相器电路的短路电流

短路功耗 P_{short}的计算公式为：$P_{short}=V_{DD}T_rQ_x$。其中，V_{DD}为供电电压，T_r 为翻转频率，Q_x 为一次翻转过程中从电源流到地的电荷量。

动态功耗主要跟电源的供电电压、翻转频率和负载电容有关。

12.1.2　静态功耗

静态功耗主要是漏电流引起的功耗，CMOS 电路漏电流下图 12-3 所示，漏电流有下面几个部分组成：

- PN 结反向电流 I_1；
- 源极和漏极之间的亚阈值漏电流 I_2；
- 栅极漏电流，包括栅极和漏极之间的感应漏电流 I_3；

- 栅极和衬底之间的隧道漏电流 I_4。

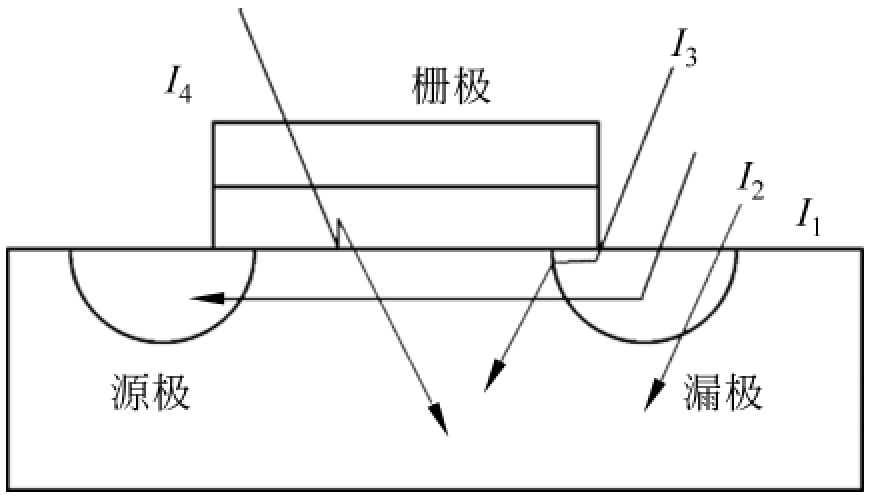


图 12-3　漏电流示意

静态功耗往往与工艺有关,静态功耗的计算公式为 $P_{peak}=V_{DD}I_{peak}$,其中,I_{peak} 为泄漏电流。

处理器的总功耗中,动态功耗特别是开关功耗占据了约 80%的来源,因此对于低功耗设计,主要示降低开关功耗,即调整处理器工作电压、工作频率。

12.2　STM32L1 系列处理器低功耗设计

12.2.1　STM32 的电源系统

STM32L1 系列处理器支持的 VDD 供电电压范围为 1.8～3.6V,如果不支持 BOR (brown out reset)欠压复位,则要求 VDD 供电范围为 1.65～3.6V。上节的功耗分析我们得知,电压越小,开关功耗越小,因此在低功耗模式下,尽量要使用低电压。

为了方便进行电源管理,STM32L1 系列把它的外设、内核等模块根据功能划分了供电区域,如图 12-4 所示,并集成了一个线性稳压器为内部提供 1.2～1.8V 的电压,供电模块包括:

1) V_{DD}

V_{DD}为外部 I/O 和内部稳压器提供电压,如果 BOR 无效,V_{DD}电压范围为 1.65～3.6V。

2) Vcore

Vcore=1.2～1.8V,Vcore 为数字外围设备、SRAM 和 Flash 提供电压。由内部稳压器产生,根据不同功耗状态,Vcore 范围是可选的(与 V_{DD}相关)。

3) V_{DDA}

V_{DDA}为外围模拟设备 ADC、DAC、复位模块、RC 振荡器和 PLL 提供电压,如果 BOR 无效,V_{DDA}电压范围为 1.65～3.6V。使用 ADC 时,V_{DDA}电压不能小于 1.8V。

4) V_{REF}－,V_{ERF}＋

V_{REF}＋为输入参考电压。只有在 LQFP144,UFBGA132,LQFP100,UFBGA100 和 TFBGA64 封装的才是有效的,其他封装情况下,默认连接到 V_{SSA}和 V_{DDA}上。

5) V_{LCD}

V_{LCD}=2.5～3.6V。LCD 控制器可由 V_{LCD}外部接口或者内部嵌入式升压转换器提供电压。

STM32L1 系列处理器的电源分配特点为：

1) 独立的 AD 和 DAC 转换器供给和参考电压

为了提高转换精度，ADC 和 DAC 有独立的电压供给，可以对数字电源滤波和隔离提供单独的 V_{DDA}和 V_{SSA}为 ADC 提供电压。

2) 独立的 LCD 供给电压

V_{LCD}可用于控制 LCD 的对比度。该电源可以采用外部电路供电，要求电压范围应为 2.56～3.6V，可以与 V_{DD}无关，也可以连接到一个外部电容上，用于 MCU 的升压转换器软件控制 LCD 的电压。

3) 电压调整器

线性电压调整器可用于所有数字电路(除了待机中的电路)。调整器的输出电压(Vcore)1.2～1.8V，可以通过编程设置为三种不同的范围。复位后，电压调整器被使能。可以配置为三种模式：主电压调节器模式 MR，低功耗运行 LPR 和待机模式。

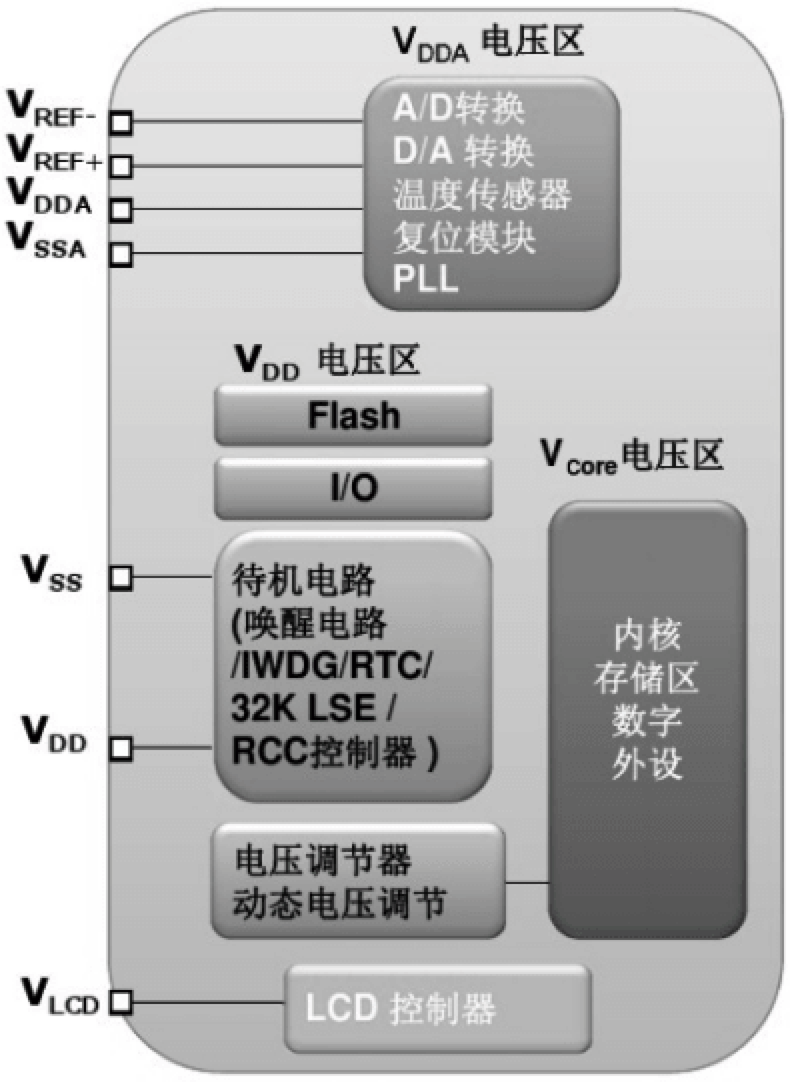


图 12-4　电源分配

- 在运行模式下，调整器处于主模式(MR)，并为 Vcore 提供全功率供电(处理器核心、存储器、数字外设)；
- 在低功耗运行模式下，调整器处于低功耗模式(LWR)，并为 Vcore 提供供电，维持寄存器及内部 SRAM 数据；
- 在睡眠模式下，调整器处于主模式(MR)，并为 Vcore domain 提供供电，维持寄存器及内部 SRAM 的数据；
- 在低功耗睡眠模式下，调整器处于低功耗模式(LWR)，并未 Vcore 提供供电，维持寄存器及内部 SRAM 数据；
- 在待机模式下，调整器处于关闭状态，除了待机电路之后，寄存器及 SRAM 中的数据都会丢失。

12.2.2　动态电压调节管理

动态电压调节管理是一项电源管理技术，根据不同的情况，增加或减少 Vcore 的电压。提高设备的性能或降低设备功耗。

根据应用情况可将处理器的电压范围分为三种：Range1、Range2 和 Range3。

(1) Range1 为高性能范围。电压调节器输出 1.8V 电压(V_{DD}电压为 2.0V)。在该范围

内，Flash 编程和擦除操作均可操作。

（2）Range2 为中等性能范围，电压调节器输出 1.5V，Flash 存储器有效，但读取时间为中等，可以编程和擦除 Flash。

（3）Range3 为低性能范围，电压调节器输出 1.2V，Flash 存储器有效，但读取时间较慢，不可以编程和擦除 Flash。

三种区间的性能如表 12-1 所示，CPU 性能、V_{DD}与 Vcore 之间的关系如图 12-5 所示。

表 12-1　不同电压范围的性能

CPU 性能	功耗	电压范围	电压值/V	最高频率/MHz	V_{DD}
高	高	1	1.8	32	2.0～3.6
中	中	2	1.5	16	1.65～3.6
低	低	3	1.2	4	1.65～3.6

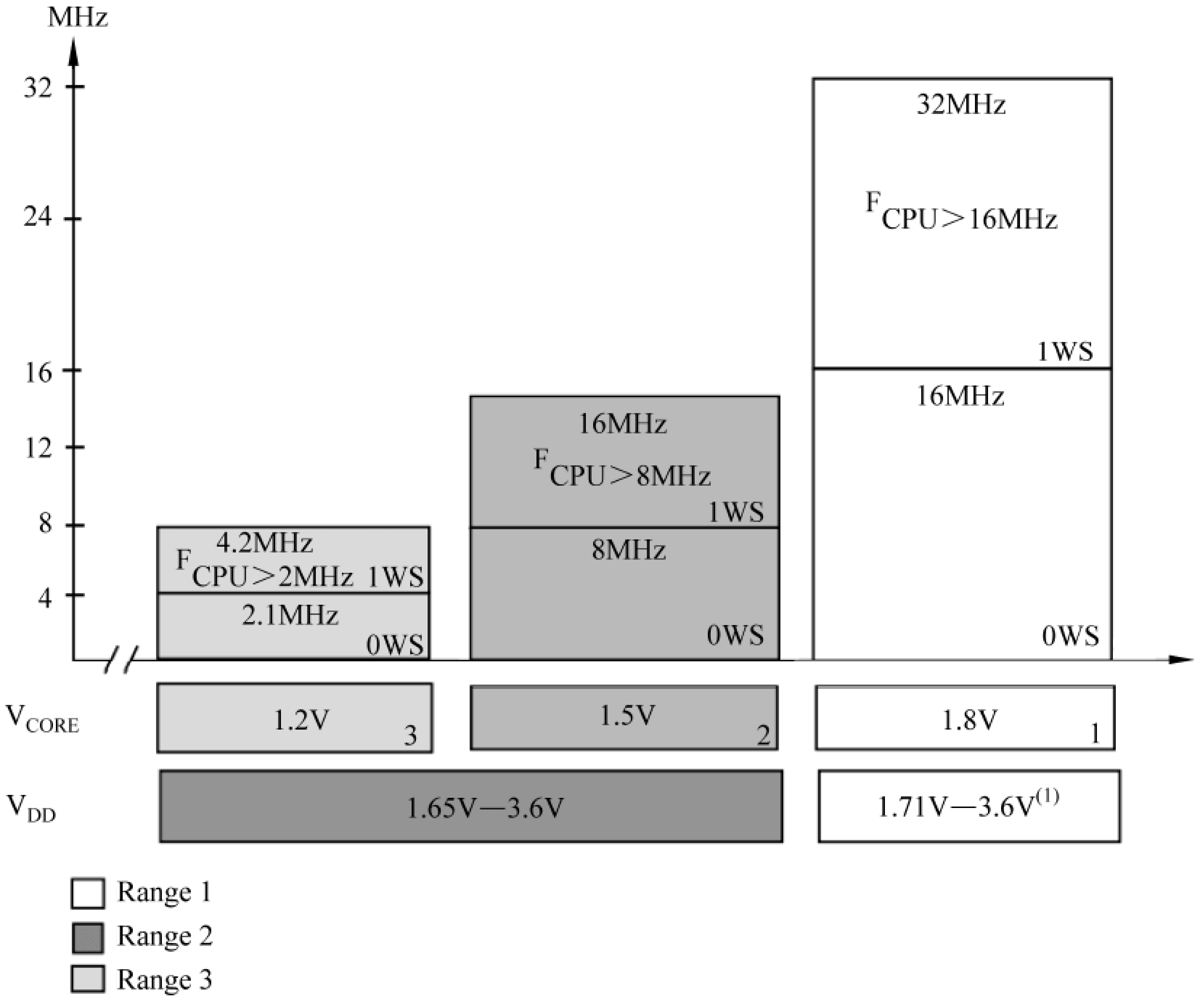


图 12-5　V_{DD}、Vcore 和处理器性能的关系

动态电压调节配置的流程如下：

- 检测 V_{DD}以确认可使用哪些 ranges；
- 轮询 PWR_CSR 寄存器的 VOSF 位直到该位为 0；
- 通过配置 PWR_CR 寄存器的 VOS[12：11]位来配置电压范围；
- 轮询 VOSF 位直到该位为 0。

12.2.3 电源检测

STM32L1xx内部集成了上电复位(POR)/掉电复位(PDR)、欠压复位(BOR)电路。当设备工作在1.8～3.6V时,BOR默认情况下是使能的,当V_{DD}电压降低到了1.8V的阈值时,将引起复位。可以通过修改阈值或者关闭BOR使得V_{DD}的最小值为1.65V。

BOR的5种阈值可以通过选择字节进行配置。为了减少待机模式下的功耗,内部电压参考VERFINT可以自动关闭。当V_{DD}低于指定的阈值时(VPOR,VPDR,VBOR),设备保持在复位模式,并且无需任何外部复位电路。

STM32L1xx内嵌可编程的电压检测器,用于监测V_{DD}/V_{DDA}的电压和跟VPVD阈值进行比较。可以选择7种不同的PVD。当V_{DD}/V_{DDA}低于或者高于VPVD阈值的时候会产生一个中断,中断可以例行的可以产生警告信息或者令MCU进入到安全的状态中,需要通过应用来使能PVD。

1) 上电复位(POR)/掉电复位(PDR)

当上电时,V_{DD}/V_{DDA}低于一个特定的阈值VPOR时,设备保持在复位状态,且不需要多余的外部复位电路。POR默认是使能的,默认阈值为1.5V。V_{DD}下降到低于VPDR阈值时,PDR让设备保持在复位状态。PDR也是默认使能的,阈值为1.5V。只有当BOR是无效时才可使用POR和PDR。

2) 欠压复位(BOR)

当设备工作在1.65～3.6V时,BOR无效,电压检测用POR/PDR。当设备工作在1.8～3.6V时,BOR在上电后使能,且其阈值VBOR为1.8V。VBOR可以通过选项字节进行修改,默认情况下为Level 4:

- BOR Level 0(VBOR0):复位阈值为:1.69～1.80 V。
- BOR Level 1(VBOR1):复位阈值为:1.94～2.1 V。
- BOR Level 2(VBOR2):复位阈值为:2.3～2.49 V。
- BOR Level 3(VBOR3):复位阈值为:2.54～2.74 V。
- BOR Level 4(VBOR4):复位阈值为:2.77～3.0 V。

当V_{DD}下降到低于所选择的VBOR阈值,则会产生复位。当V_{DD}上升到高于VBOR上限时,则会释放复位,设备启动。

3) 可编程电压检测器(PVD)

可以使用PVD来检测VDD电源,并与PWR_CR寄存器PLS[2:0]定义的阈值进行比较,通过PWR_CSR寄存器的PVODO标志识别VDD是低于还是高于PVD阈值。该事件连接到了外部中断控制器的EXTI16,若配置了EXTI寄存器则可产生中断。当V_{DD}上升到高于PVD阈值或者下降到低于PVD阈值时(与EXTI16上的边缘配置有关)会产生中断。可应用于通过中断服务处理紧急的关机任务。

4) 内部参考电压(VERFINT)

内部参考电压的功耗并不是可忽略的,为了减少功耗,可以通过PWR_CR寄存器的

ULP 位令内部参考电压失效。

12.2.4　低功耗模式

系统启动后默认处于运行模式，CPU 时钟由 HCLK 驱动。当 CPU 不需要保持在运行模式下时，可以进入多种低功耗模式。STM32L1 系列提供了五种低功耗模式：

- 低功耗运行模式：电压调节器处于低功耗模式，限制了时钟频率和可运行外设数量；
- 睡眠模式：Cortex-M3 停止，外设保持运行；
- 低功耗睡眠模式：Cortex-M3 停止，限制了时钟频率和可运行外设数量，电压调节器处于低功耗模式，RAM 掉电，Flash 停止；
- 停止模式：所有时钟停止，电压调节器继续运行，并处于低功耗模式；
- 待机模式：Vcore 掉电关机。

另外，可以通过减少系统时钟频率、关闭不使用的外设时钟减少运行时功耗。

各种低功耗模式的进入和换序条件如图 12-6 所示，不同低功耗模式的电流消耗如图 12-7 所示。

模式	进入	唤醒	对VCORE域时钟影响	对VDD域时钟影响	内部电源变换器	I/O 状态	唤醒延迟
低功耗运行模式	设置LPSDSR 位和LPRUN 位 + 设置系统时钟	恢复系统时钟和内部电源变换器的工作模式	无	无	低功耗模式	所有I/O口的状态和运行时保持一致	无
睡眠模式	WFI	任意中断	CPU时钟关闭，不影响其他时钟	无	正常模式		无
	WFE	唤醒事件					
低功耗睡眠模式	设置LPSDSR 位 + WFI	任意中断		无	低功耗模式		电源变换器的改变时间+FLASH的唤醒时间
	设置LPSDSR 位 + WFE	唤醒事件					
停止模式	设置PDDS和LPSDSR位 + 设置SLEEPDEEP位 + WFI 或 WFE	任意EXTI中断（在EXTI寄存器中配置的内部或者外部中断源）	所有的VCORE域时钟都关闭	HSI，HSE和MSI都关闭	正常模式/低功耗模式		MSI RC的唤醒时间+电源变换器的唤醒时间 + FLASH的唤醒时间 唤醒的典型值为7.9us
待机模式	设置PDDS和 + 设置SLEEPDEEP位 + WFI 或 WFE	WKUP引脚的上升沿，RTC报警（Alarm A 或者 Alarm B），RTC唤醒事件，RTC时间戳事件，RTC侵入事件，NRST引脚的外部复位，IWDG复位			关闭	所有I/O口都保持在高阻状态	V_{REFINT} 开的情况下唤醒典型值为57.2us V_{REFINT} 关的情况下唤醒典型值为2.4ms

图 12-6　不同模式的进入和唤醒条件

1. 低功耗模式下的时钟

1）睡眠和低功耗睡眠模式

在睡眠和低功耗睡眠模式下，CPU 的时钟处于停止状态。寄存器接口时钟和所有外设

模式	STM32L15x典型值	STM32F10x 典型值
运行模式功耗 代码在Flash中运行，内核供电范围选择3，开外设时钟	230μA/MHz	-
运行模式功耗 代码在RAM中运行，内核供电范围选择3，开外设时钟	186μA/MHz	-
低功耗运行模式功耗 代码在RAM中运行，使用内部RC(32kHz的MSI)，开外设时钟	10.4μA	-
睡眠模式功耗 代码在Flash中运行，主时钟频率为16 MHz，关所有外设时钟	650μA	-
睡眠模式功耗 代码在Flash中运行，主时钟频率为16 MHz，开所有外设时钟	2.5mA	-
低功耗睡眠模式功耗 代码在Flash中运行，主时钟频率为32kHz, 内部电源变换器工作在低功耗模式下，运行一个32kHz的定时器	6.1μA	-
停止模式功耗 内部电源变换器工作在低功耗模式下，关闭低速/高速内部振荡器和高速外部振荡器，不使能独立看门狗	0.43μA w/o RTC 1.3μA w/ RTC	14 μA
待机模式功耗 使用低速内部振荡器，不使能独立看门狗，关闭RTC	0.27μA	2 μA
待机模式功耗 使能RTC	1.0 μA	-

图 12-7　不同模式的电流消耗及其和 F10x 系列的对比

时钟均可通过软件进行关闭。当处于低功耗睡眠模式时，RAM 接口时钟处于掉电状态。AHB 总线到 APB 总线的桥时钟可以通过硬件进行关闭。

2）停止和待机模式

在停止和待机模式下，系统时钟和所有高速时钟均是处于停止状态：

- PLL 无效；
- 内部 RC 16MHz(HSI)振荡器无效；
- 外部 1M～24MHz(HSE)振荡器无效；
- 内部 65kHZ～4MHz(MSI)振荡器无效。

当通过中断退出停止模式或者复位退出待机模式时，内部 MSI 被选择为系统时钟。当设备退出停止模式时，之前的 MSI 配置仍是有效的。当设备退出待机模式时，被重置为默认的 2MHZ。

运行模式下，可以通过降低系统时钟和关闭外设时钟降低功耗。外设时钟受以下寄存器 RCC_AHBENR、RCC_APB2ENR、RCC_APB1ENR 控制，在睡眠模式时，为了关闭外设时钟可以通过复位 RCC_AHBLPENR 和 RCC_APBxLPENR 相应的位。

2. 低功耗运行模式

在运行时，为了进一步减少功耗，电压调整器可以配置为低功耗模式，在该模式下，系统的频率不应超过 f_MSI Range1。

1）进入低功耗模式

- 通过 RCC_APBxENR 和 RCC_AHBENR 寄存器使能或者关闭每一个数字 IP

时钟；

- 系统时钟频率需要下降到不得超过 f_MSI Range1；
- 通过置位 LPRUN 和 LPSDSR 强制让调整器运行在低功耗模式下。

2）退出低功耗模式

- 配置调节器运行在主电压调节器模式；
- 如果有需要，开启 Flash 存储器；
- 按需要增加系统时钟频率。

3. 睡眠模式

1）进入睡眠模式

通过执行 WFI 或 WEF 可以进行到睡眠模式下，根据 Cortex-M3 SCR 寄存器的 SLEEPONEXIT 位，睡眠模式的进入机制有两种：

- 立即睡眠：如果 SLEEPONEXIT 被清除，当 WFI/WEF 指令被执行时，MCU 则进入睡眠模式；
- 异常退出睡眠：如果 SLEEPONEXIT 被置位，当最低优先级的 ISR 退出时，MCU 则进入睡眠模式。

2）退出睡眠模式

- 如果是通过 WFI 进入到睡眠模式的，发生任何能被 NVIC 所确认的外设中断均会将设备从睡眠模式中唤醒；
- 如果是通过 WEF 进入到睡眠模式的，一检测到有事件发生时，设备将从睡眠模式中唤醒。

4. 低功耗睡眠模式

1）进入低功耗睡眠模式

通过配置电压调整器处于低功耗模式，并且执行 WFI/WEF 指令，则可进入到低功耗睡眠模式。在该模式下，Flash 不可用，RAM 存储器可用。在该模式下，系统时钟频道不应高于 f_MSI Range1。只有当 Vcore 处于 Range2 时，才可进入低功耗睡眠模式。

低功耗睡眠同样有立即睡眠和异常退出睡眠两种模式，进入睡眠的流程为：

- 可通过控制位关闭 Flash，减少功耗；
- 通过 RCC_APBxENR 和 RCC_AHBENR 寄存器使能或者关闭每一个数字外设时钟；
- 必须减少系统时钟频率；
- 强制电压调节器进入到低功耗模式（LPSDSR 位）；
- 执行 WFI/WEF 指令以进入到睡眠模式。

2）退出低功耗睡眠模式

- 如果是通过 WFI 进入到睡眠模式的，发生任何能被 NVIC 所确认的外设中断均会将设备从睡眠模式中唤醒；
- 如果是通过 WEF 进入到睡眠模式的，一检测到有事件发生时，设备将从睡眠模式中唤醒。

5. 停止模式

停止模式基于 Cortex-M3 深度睡眠模式，电压调节器可以配置为正常或者低功耗模式。

在停止模式时，所有位于 Vcore 的时钟停止，PLL、MSI、HSI 和 HSE 均无效，内部 SRAM 和寄存器将被保留。为了在停止模式下获取更低功耗，内部 Flash 一般配置为低功耗模式，在进入停止模式前，可关闭 VREFINT，BOR，PVD 和温度传感器。在退出停止模式时通过软件设置使之前关闭的功能重新开启。

为了进一步减少停止模式下的功耗，通过配置 PWR_CR 寄存器的 LPSDSR 位内部电压调节器可以设置为低功耗模式，停止模式下，以下功能模块可以进行单独配置：

- 独立看门狗(IWDG)：写其主要的寄存器或者硬件选项来启动 IWDG；
- 实时时钟(RTC)：配置 RCC_CSR 寄存器的 RTCEN 位；
- 内部 RC 振荡器(LSI RC)：配置 RCC_CSR 寄存器的 LSION 位；
- 外部 32.768kHZ 振荡器(LSE OSC)：配置 RCC_CSR 寄存器的 LSEON 位。

ADC，DAC 或 LCD 在停止模式也会消耗能源，最好将 ADC_CR2 寄存器的 ADON 和 DAC_CR 寄存器的 ENx 位需要配置为 0。

当退出停止模式时会产生一个中断或者唤醒事件，MSI RC 振荡器被选择为系统时钟。

6. 待机模式

待机模式下会得到最低的功耗，电压调节器关闭，Vcore 关闭，PLL、MSI、HSI 和 HSE 失效。除了 RTC 寄存器、RTC 备份寄存器，待机电路外 SRAM 和寄存器上下文均被丢失。

12.3 功耗控制寄存器

1. PWR 功耗控制寄存器 PWR_CR

PWR 功耗控制寄存器的有效域定义如图 12-8 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	LPRUN	Res.	VOS[1:0]		FWU	ULP	DBP	PLS[2:0]			PVDE	CSBF	CWUF	PDDS	LPSDSR
	rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rc_w1	rc_w1	rw	rw

图 12-8　PWR 功耗控制寄存器

LPRUN：低功耗运行模式，当 LPRUN 位与 LPSDSR 位一起置 1 时，电压调节器从主模式切换到低功耗模式。否则，它仍处于主模式。0 表示主电压电压调节器模式，1 表示电压调节器低功耗模式。

VOS[1：0]：电压调整范围选择，00 表示禁止改变电压范围，01～11 分别表示 Range1～

Range3。

FWU、ULP：快速唤醒和超低功耗模式，该位与 ULP 位配合使用。如果 ULP=0，则忽略 FWU；如果 ULP=1 且 FWU=1，则从低功耗模式退出时，将忽略 VREFINT 启动时间；如果 ULP=1 且 FWU=0，则仅在 VREFINT 准备就绪时退出低功耗模式。

DBP：禁用备份写保护，0 表示复位时禁止访问 RTC，RTC 备份和 RCC CSR 寄存器，1 表示允许。

PLS[2：0]：PVD 电平选择，000～110 分别表示 1.9V、2.1V、2.3V、2.5V、2.7V、2.9V、3.1V，111 表示 PVD 为外部输入模拟电压（内部与 VREFINT 比较），此时 PVD_IN 输入（PB7）必须配置为模拟输入。

PVDE：电源电压检测器使能，0 表示 PVD 禁用，1 表示启用。

CSBF：清除待机标志，写 1 表示清除 SBF 待机标志。

2. PWR 功耗控制/状态寄存器 PWR_CSR

PWR 功耗控制/状态寄存器的有效域定义如图 12-9 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved					EWUP3	EWUP2	EWUP1	Reserved		REGLPF	VOSF	VREFINTRDYF	PVDO	SBF	WUF
					rw	rw	rw			r	r	r	r	r	r

图 12-9 PWR 功耗控制/状态寄存器

EWUP3～EWUP1：使能 WKUP 引脚 3～1，0 表示 WKUP 引脚 3～1 用于通用 I/O，1 表示 WKUP 引脚 3～1 用于从待机模式唤醒，强制输入下拉，WKUP 引脚 3～1 的上升沿将系统从待机模式唤醒。

REGLPF：调节器 LP 标志，当 MCU 处于低功耗运行模式时，该位由硬件置 1，当 MCU 退出低功耗运行模式时，该位保持为 1 直到电压调节器进入主模式。0 表示电压调节器在主模式下准备就绪，1 表示电压调节器处于低功耗模式。

VOSF：电压调节选择标志，在更改电压范围后，内部稳压器需要一定的时间，VOSF 位表示电压调节器已达到 VOS 位定义的电压电平，1 表示未达到。

VREFINTRDYF：内部参考电压（VREFINT）就绪标志位，0 表示 VREFINT 关闭，1 表示 VREFINT 准备就绪。

PVDO：PVD 输出，该位由硬件置 1 和清除。仅当 PVDE 位使能 PVD 时有效，0 表示 VDD 高于 PLS[2：0]位选择的 PVD 阈值，1 表示低于 PVD 阈值。

SBF：待机标志，该位由硬件置 1，仅由 POR/PDR 清除，或者通过设置 PWR 功率控制寄存器（PWR_CR）中的 CSBF 位置 1，0 表示设备未处于待机模式，1 表示处于待机模式。

WUF：唤醒标志，该位由硬件置 1，并由系统复位或通过设置 CWUF 位清零，0 表示没有发生唤醒事件，1 表示收到唤醒事件。

12.4　PWR 寄存器结构及库函数

12.4.1　PWR 寄存器结构

PWR 寄存器结构，PWR_TypeDeff 在文件 stm32l1xx.h 中定义如下：

```
typedef struct
{
  __IO uint32_t CR;
  __IO uint32_t CSR;
} PWR_TypeDef;
```

ADC 外设声明于文件 stm32l1xx.h：

```
#define PERIPH_BASE                 ((uint32_t)0x40000000)
#define APB1PERIPH_BASE PERIPH_BASE
#define APB2PERIPH_BASE             (PERIPH_BASE + 0x10000)
#define AHBPERIPH_BASE              (PERIPH_BASE + 0x20000)
#define PWR_BASE                    (APB1PERIPH_BASE + 0x7000)
#define PWR                         ((PWR_TypeDef *) PWR_BASE)
```

12.4.2　PWR 库函数

PWR 库函数如表 12-2 所示。

表 12-2　PWR 库函数

函　　数	功　　能
PWR_DeInit	PWR 寄存器复位
PWR_RTCAccessCmd	RTC 备份写保护启用或禁用
PWR_PVDLevelConfig	PVD 参考电压设置
PWR_PVDCmd	PVD 启用或禁用
PWR_WakeUpPinCmd	启用或禁用唤醒引脚功能
PWR_FastWakeUpCmd	低功耗模式快速唤醒使能
PWR_UltraLowPowerCmd	启用或禁用内部参考电源低功耗模式
PWR_VoltageScalingConfig	设置 Vcore 电压范围
PWR_EnterLowPowerRunMode	进入低功耗运行模式

续表

函　　数	功　　能
PWR_EnterSleepMode	进入休眠模式
PWR_EnterSTOPMode	进入停止模式
PWR_EnterSTANDBYMode	进入待机模式
PWR_GetFlagStatus	获取状态寄存器
PWR_ClearFlag	清除状态位

1）函数 PWR_RTCAccessCmd

函数功能：启用或禁用对 RTC 和备份寄存器的访问。

函数原型：void PWR_RTCAccessCmd(FunctionalState NewState)。

输入参数 NewState：是否允许访问，取值范围为 ENABLE 或 DISABLE。

2）函数 PWR_RTCAccessCmd

函数功能：配置电源电压检测器(PVD)的电压阈值。

函数原型：void PWR_PVDLevelConfig(uint32_t PWR_PVDLevel)。

输入参数 PWR_PVDLevel：PVD 电压值，取值范围为：

- PWR_PVDLevel_0：PVD 检测电平设置为 1.9V。
- PWR_PVDLevel_1：PVD 检测电平设置为 2.1V。
- PWR_PVDLevel_2：PVD 检测电平设置为 2.3V。
- PWR_PVDLevel_3：PVD 检测电平设置为 2.5V。
- PWR_PVDLevel_4：PVD 检测电平设置为 2.7V。
- PWR_PVDLevel_5：PVD 检测电平设置为 2.9V。
- PWR_PVDLevel_6：PVD 检测电平设置为 3.1V。
- PWR_PVDLevel_7：外部输入模拟电压(内部比较)。

3）函数 PWR_PVDCmd

函数功能：启用或禁止电源电压检测器(PVD)。

函数原型：void PWR_PVDCmd(FunctionalState NewState)。

输入参数 NewState：是否启用 PVD，取值为 ENABLE 或 DISABLE。

4）函数 PWR_WakeUpPinCmd

函数功能：启用或禁用 I/O 唤醒功能。

函数原型：void PWR_WakeUpPinCmd(uint32_t PWR_WakeUpPin, FunctionalState NewState)。

输入参数 PWR_WakeUpPin：指定要配置的 I/O 引脚，取值为 PWR_WakeUpPin_x(x=1,2,3)。

输入参数 NewState：是否启用 I/O 作为唤醒引脚，取值为 ENABLE 或 DISABLE。

5）函数 PWR_WakeUpPinCmd

函数功能：启用或禁用 I/O 唤醒功能。

函数原型：void PWR_WakeUpPinCmd(uint32_t PWR_WakeUpPin，FunctionalState NewState)。

输入参数 PWR_WakeUpPin：指定要配置的 I/O 引脚，取值为 PWR_WakeUpPin_x(x=1，2，3)。

输入参数 NewState：是否启用 I/O 作为唤醒引脚，取值为 ENABLE 或 DISABLE。

6）函数 PWR_VoltageScalingConfig

函数功能：配置电压调节器电压范围。

函数原型：void PWR_VoltageScalingConfig(uint32_t PWR_VoltageScaling)。

输入参数 PWR_VoltageScaling：指定电压范围，取值为 PWR_VoltageScaling_Rangex (x=1，2，3)。

7）函数 PWR_EnterLowPowerRunMode

函数功能：是否进入低功耗运行模式。

函数原型：void PWR_EnterLowPowerRunMode(FunctionalState NewState)。

输入参数 NewState：是否进入低功耗运行模式，取值为 ENABLE 或 DISABLE。

8）函数 PWR_EnterSleepMode

函数功能：是否进入进入睡眠模式。

函数原型：void PWR_EnterSleepMode(uint32_t PWR_Regulator，uint8_t PWR_SLEEPEntry)。

输入参数 PWR_Regulator：休眠模式下的电压调节器状态，取值为：

- PWR_Regulator_ON：电压调节器开启；
- PWR_Regulator_LowPower：电压调节器低功耗模式。

输入参数 PWR_SLEEPEntry：指定是否使用 WFI 或 WFE 指令进入 SLEEP 模式。其取值为：

- PWR_SLEEPEntry_WFI：使用 WFI 指令进入休眠模式；
- PWR_SLEEPEntry_WFE：使用 WFE 指令进入休眠模式。

9）函数 PWR_EnterSTOPMode

函数功能：是否进入进入停止模式。

函数原型：void PWR_EnterSTOPMode(uint32_t PWR_Regulator，uint8_t PWR_STOPEntry)。

输入参数 PWR_Regulator：停止模式下的电压调节器状态，取值与 PWR_EnterSleepMode 函数的参数相同。

输入参数 PWR_STOPEntry：指定是否使用 WFI 或 WFE 指令进入 STOP 模式。其取值为：

- PWR_STOPEntry_WFI：使用 WFI 指令进入休眠模式；
- PWR_STOPEntry_WFE：使用 WFE 指令进入休眠模式。

10）函数 PWR_EnterSTANDBYMode

函数功能：是否进入待机模式。

函数原型：void PWR_EnterSTANDBYMode(void)。

11）函数 PWR_GetFlagStatus

函数功能：检查是否设置了指定的 PWR 标志。

函数原型：FlagStatus PWR_GetFlagStatus(uint32_t PWR_FLAG)。

输入参数 PWR_FLAG：指定要检查的标志，取值为：

- PWR_FLAG_WU：唤醒标志；
- PWR_FLAG_SB：待机标志；
- PWR_FLAG_PVDO：PVD 输出；
- PWR_FLAG_VREFINTRDY：内部电压参考就绪标志；
- PWR_FLAG_VOS：电压调节选择标志；
- PWR_FLAG_REGLP：电压调节器低功耗模式标志。

返回值 PWR_FLAG 为状态值，取值为 SET 或 RESET。

12）函数 PWR_GetFlagStatus

函数功能：清除指定的 PWR 标志。

函数原型：void PWR_ClearFlag(uint32_t PWR_FLAG)。

输入参数 PWR_FLAG：指定要清除的标志，取值为：

- PWR_FLAG_WU：唤醒标志；
- PWR_FLAG_SB：待机标志；

12.5 PWR 案例

【例 12-1】 寄存器级操作案例。

```
PWR_EnterSTOPMode(uint32_t PWR_Regulator, uint8_t PWR_STOPEntry)
{
  uint32_t tmpreg = 0;
  //配置 STOP 模式下的电压调节器
  tmpreg = PWR->CR;
  tmpreg &= CR_DS_MASK;                 //清除 PDDS 和 LPDSR 域
  tmpreg |= PWR_Regulator;              //根据输入的 PWR_Regulator 设置 LPDSR 域
  PWR->CR = tmpreg;                     //配置值写回到控制寄存器 CR
  /* Set bit of Cortex System Control Register */
  SCB->SCR |= SCB_SCR_SLEEPDEEP;        //配置 Cortex-M3 系统控制寄存器的 SLEEPDEEP 域
  if(PWR_STOPEntry == PWR_STOPEntry_WFI)  //选择停止模式的进入条件为 WFI
  {
    __WFI();                            //休眠，等待中断唤醒
```

```
    }
    Else                            //进入条件为 WFE
    {
        __WFE();                    //休眠,等待事件发生
    }
    //清除 Cortex-M3 系统控制寄存器的 SLEEPDEEP 域
    SCB->SCR &= (uint32_t)~((uint32_t)SCB_SCR_SLEEPDEEP);
}
```

【例 12-2】 基于库函数的低功耗运行模式配置。

```
void LowPowerRunMode_Measure(void)
{
    //系统时钟 MSI Range0 (65kHz)
    RCC_DeInit();                                   //RCC 复位
    FLASH_SetLatency(FLASH_Latency_0);              //Flash 0 等待
    FLASH_PrefetchBufferCmd(DISABLE);               //禁止预取
    FLASH_ReadAccess64Cmd(DISABLE);                 //禁止 64 位访问
    //使能 PWR APB1 时钟
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR, ENABLE);
    //选择电压调节器为 Range2 (1.5V)
    PWR_VoltageScalingConfig(PWR_VoltageScaling_Range2);
    //等待电压调节器稳定
    while(PWR_GetFlagStatus(PWR_FLAG_VOS) != RESET);
    RCC_HCLKConfig(RCC_SYSCLK_Div2);                // HCLK = SYSCLK/2 = 32kHz
    RCC_PCLK2Config(RCC_HCLK_Div1);                 // PCLK2 = HCLK
    RCC_PCLK1Config(RCC_HCLK_Div1);                 // PCLK1 = HCLK
    RCC_MSIRangeConfig(RCC_MSIRange_0);             // MSI 设置为 65.536kHz
    RCC_SYSCLKConfig(RCC_SYSCLKSource_MSI);         // MSI 作为系统时钟源
    while (RCC_GetSYSCLKSource() != 0x00) ;
    //配置所有的 GPIO 为模拟,减少功耗
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA | RCC_AHBPeriph_GPIOB |
    RCC_AHBPeriph_GPIOC | RCC_AHBPeriph_GPIOD | RCC_AHBPeriph_GPIOE |
    RCC_AHBPeriph_GPIOH | RCC_AHBPeriph_GPIOF | RCC_AHBPeriph_GPIOG, ENABLE);
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_All;
    GPIO_Init(GPIOC, &GPIO_InitStructure);
    GPIO_Init(GPIOD, &GPIO_InitStructure);
    GPIO_Init(GPIOE, &GPIO_InitStructure);
    GPIO_Init(GPIOH, &GPIO_InitStructure);
    GPIO_Init(GPIOF, &GPIO_InitStructure);
    GPIO_Init(GPIOG, &GPIO_InitStructure);
```

```
    GPIO_Init(GPIOA, &GPIO_InitStructure);
    GPIO_Init(GPIOB, &GPIO_InitStructure);
    //关闭 GPIO 时钟

    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA | RCC_AHBPeriph_GPIOB |
    RCC_AHBPeriph_GPIOC | RCC_AHBPeriph_GPIOD | RCC_AHBPeriph_GPIOE |
    RCC_AHBPeriph_GPIOH | RCC_AHBPeriph_GPIOF | RCC_AHBPeriph_GPIOG, DISABLE);
    //进入低功耗运行模式
    PWR_EnterLowPowerRunMode(ENABLE);
    while(PWR_GetFlagStatus(PWR_FLAG_REGLP) == RESET);
    //此时进入 LP 模式
    //退出 LP RUN 模式
    PWR_EnterLowPowerRunMode(DISABLE);
    while(PWR_GetFlagStatus(PWR_FLAG_REGLP) != RESET);
}
```

参 考 文 献

[1] Joseph Yiu. Cortex-M3 权威指南[M]. 宋岩,译. 北京：北京航空航天大学出版社,2009.

[2] 陈泽宇. 计算机组成与系统结构[M]. 北京：清华大学出版社,2009.

[3] Stmicroelectronics. RM0038：STM32L100xx，STM32L151xx，STM32L152xx and STM32L162xx advanced ARM-based 32-bit MCUs[EB/OL]. (2017-09-04)[2018-02-01]. https：//www. st. com/content/ccc/resource/technical/document/reference_manual/cc/f9/93/b2/f0/82/42/57/CD00240193. pdf/files/CD00240193. pdf/jcr：content/translations/en. CD00240193. pdf.

[4] Stmicroelectronics. PM0056：STM32F10xxx/20xxx/21xxx/L1xxxx Cortex-M3 programming manual [EB/OL]. (2017-12-20)[2018-02-01]. https：//www. st. com/content/ccc/resource/technical/document/programming _ manual/5b/ca/8d/83/56/7f/40/08/CD00228163. pdf/files/CD00228163. pdf/jcr：content/translations/en. CD00228163. pdf.

[5] ARM. Cortex-M3 Devices Generic User Guider[EB/OL]. [2018-02-01]. http：//infocenter. arm. com/help/topic/com. arm. doc. dui0552a/DUI0552A_cortex_m3_dgug. pdf.

清华科技大讲堂
前沿·科技·分享

算法竞赛

入门到进阶

- 算法思路　一点就透，豁然开朗
- 模板代码　结构精巧，清晰易读
- 知识体系　由浅入深，逐步推进
- 赛事相关　参赛秘籍，高手经验

微课版
120分钟视频讲解

罗勇军 郭卫斌 ◎ 著

清华大学出版社

清华科技大讲堂

算法竞赛入门到进阶

罗勇军　郭卫斌　著

清华大学出版社
北　京

内 容 简 介

本书是算法竞赛的入门和进阶教材，包括算法思路、模板代码、知识体系、赛事相关等内容。本书把竞赛常用的知识点和竞赛题结合起来，讲解清晰、透彻，帮助初学者建立自信心，快速从实际问题入手，模仿经典代码解决问题，进入中级学习阶段。

全书分为12章，覆盖了目前算法竞赛中的主要内容，包括算法竞赛概述、算法复杂度、STL和基本数据结构、搜索技术、高级数据结构、基础算法思想、动态规划、数学、字符串、图论、计算几何。

本书适合用于高等院校开展的ICPC、CCPC等算法竞赛培训，中学NOI信息学竞赛培训，以及需要学习算法、提高计算思维的计算机工作者。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989　13701121933

图书在版编目(CIP)数据

算法竞赛入门到进阶/罗勇军，郭卫斌著. —北京：清华大学出版社，2019
(清华科技大讲堂)
ISBN 978-7-302-52915-6

Ⅰ.①算…　Ⅱ.①罗…　②郭…　Ⅲ.①计算机算法　Ⅳ.①TP301.6

中国版本图书馆CIP数据核字(2019)第083538号

策划编辑：魏江江
责任编辑：王冰飞
封面设计：刘　键
责任校对：李建庄
责任印制：杨　艳

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦A座　**邮　　编**：100084
社 总 机：010-62770175　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 刷 者：北京富博印刷有限公司
刷 订 者：北京市密云县京文制本装订厂
经　　销：全国新华书店
开　　本：185mm×260mm　**印　　张**：22.5　**字　　数**：546千字
版　　次：2019年8月第1版　**印　　次**：2019年8月第1次印刷
印　　数：1～2500
定　　价：59.80元

产品编号：081639-01

推　荐　序

大学生算法竞赛，例如 ICPC（国际大学生程序设计竞赛）、CCPC（中国大学生程序设计竞赛），是目前中国最具影响力的大学生计算机赛事。

这些算法竞赛之所以具有很大的影响力，是因为它们综合考察了参赛队员在编码能力、算法知识、逻辑思维、创新能力、团队合作等各方面的素质。很多从算法竞赛中走出的获奖学生已陆续成长为杰出的软件工程师。近年来，在国内、国际闻名的 IT 公司中，有算法竞赛背景的创业者也层出不穷。

我长期从事算法竞赛的培训工作，深刻认识到算法竞赛在我国 IT 教育中的关键作用。可以说，IT 行业决定了一个国家的未来发展高度。目前，中国的 IT 行业已经在世界上颇具影响力。中国每年培养约 100 万信息类大学生，但仍然供不应求。特别是高级软件工程师，更是各大企业、创业公司争抢的对象。算法竞赛的目标正是培养杰出的软件工程师，并且中国 20 多年的算法竞赛历史已经证明，算法竞赛培训是非常有效的手段。

随着算法竞赛的发展，全国大部分高校陆续开展了算法竞赛的课内或课外课程，各大学的竞赛指导老师为之付出了极大的心血。

算法竞赛不同于其他的学科竞赛，它的长期性、艰苦性使它成为一个高难度的学习活动。参赛队员需要长期持之以恒地学习、训练，指导老师也需要在竞赛培训、教材编写、日常管理、组织参赛、主办比赛、维护 OJ 系统等方面做大量琐碎而艰苦的工作。竞赛教材的编写是其中一项重要内容，近些年，国内也陆续出版了十几种相关教材，这些都是算法竞赛学生日常学习的基本资料。

我的同行罗勇军老师是华东理工大学的竞赛教练，长期从事算法竞赛的指导工作。我和罗老师很早就认识，虽然不常见面，但网络交流很频繁，经常就竞赛学生的有效管理模式进行经验交流。

令我印象深刻的是，2008 年我校（杭州电子科技大学）主办了 ACM-ICPC 亚洲区域赛，罗老师带领的参赛队一举获得金牌，并因此入围了次年于瑞典举办的第 33 届 ACM-ICPC World Final，这让当时的我既羡慕，又深受鼓舞。一年后，我校也实现了 Final 的历史性突破，并在接下来的几年中先后 5 次入围全球总决赛。

罗老师作为一名十几年坚持在教学第一线的金牌教练，在总结长期的算法竞赛教学经验的基础上写下了《算法竞赛入门到进阶》一书。通读下来，这本书的语言描述贴近学生的阅读习惯，有很好的可读性，对基础算法的讲解详细、清晰、通俗易懂；仔细读之，读者能够深刻感受到作者为之付出的心血。

书中的例题大部分来自杭州电子科技大学在线提交系统（HDOJ），这让我倍感亲切。

同时我想，这对于国内的读者来说也一定是乐于看到的，毕竟绝大部分参加竞赛的学生都是HDOJ 的用户。

总之，希望罗老师的这本《算法竞赛入门到进阶》能给广大的参赛学生带来实实在在的帮助。

刘春英
杭州电子科技大学
2019 年 6 月

前　言

算法竞赛，例如 ACM-ICPC、CCPC 等，在中国已经活跃多年，是最具影响力的大学生计算机竞赛。目前，已经出版的算法竞赛书也有 30 多部，有一些被队员们奉为“宝书”，有很好的口碑。本书作者是竞赛教练，因为工作的原因，详细阅读过这些书。这些书，或者讲解深刻让人佩服，或者娓娓道来令人愉悦，或者洋洋大观让人欲罢不能。读经典书，甘之如饴。

在多年的竞赛教练工作中，本书作者作为喜欢自我表现的社会人，也常常跃跃欲试，试图写出一本新的经典书。本书作者认为，竞赛队员在算法竞赛学习中的痛点需求如下。

算法思路：一点就透，豁然开朗。

模板代码：结构精巧，清晰易读。

知识体系：由浅入深，逐步推进。

赛事相关：参赛秘籍，高手经验。

上面立的几个 flag 虽然高不可攀，但确实是本书作者内心的旗帜。

本书是一本“竞赛书”，不是计算机算法教材，也不是编程语言书，因此对大多数知识点本身不会做过多的讲解，而是把重点放在讲解竞赛所常用的知识点上，以及如何把知识点和竞赛题结合起来。当然，由于编程竞赛涉及太多知识点，一本竞赛书不可能面面俱到，把所有内容都堆砌进来。市面上还有太多经典的算法教材和编程语言教材，这都是竞赛队员应该认真阅读的。

本书对知识点进行了精心的剖析。很多知识点看起来复杂难解，但如果结合清晰的代码、生动的文字、通俗的比喻、一目了然的图解、画龙点睛的注解，就能让人豁然开朗。这也是本书的目标。

代码能力体现了编程者的实力。学习别人的好代码是提高自己编码水平的捷径。本书把知识点讲解和竞赛题目紧密地结合在一起，同时给出实用的代码。这些代码有的是作者精心组织和编写的，有的是搜索大量资料后进行整理总结的结果。其中很多代码完全可以作为编程的模板，希望能对参赛学生起到参考的作用。特别是经典问题，往往有经典代码，凝结了很多人的劳动。本书作者并没有独创经典代码的能力，因此书中不可避免地引用和改写了一些公开的代码。对于一些能找到出处的经典代码，在书中都标注了出处。

本书主要面向初学者和中级进阶者。初学者面对海量繁杂的竞赛知识点往往会产生深深的无力感和挫折感，本书由浅入深地讲解知识点，逐步推进，帮助初学者建立自信心，从而快速地从能理解的实际问题入手，模仿经典代码解决问题，进入中级学习阶段。

竞赛是很专业的活动，经验非常重要。书中就一些日常训练和参赛的细节问题介绍了作者的体会。

学习算法竞赛有很大难度，需要精通编程语言、掌握很多算法，但是这并不意味着需要先学好算法和编程语言才能进行竞赛训练。事实上，建议初学者从零基础就开始学习算法

编程竞赛，与算法学习和语言学习同步进行。竞赛是操练的擂台，竞赛题目把知识点和具体问题结合起来，让学到的知识有了打击的"力点"。

以上是本书的特点，希望本书能给算法竞赛的初学者和进阶学习者以较大的帮助。如果是初学者，通过本书可以快速入门，例如了解竞赛的知识点、建立算法思维、动手写出高效率的代码。如果是中级进阶者，学习本书，可以更透彻地掌握复杂算法的思想、学习经典代码、完善知识体系，从而更自信地加入到竞争激烈的比赛活动中。

本书提供教学大纲、教学课件、程序源码，扫描封底的课件二维码可以下载；本书还提供 120 分钟的视频讲解，扫描书中的二维码可以在线观看。

在本书的编写过程中，华东理工大学竞赛队员提出了一些建议，感谢 2015 级队长姚远，以及王亦凡、王泽宸、翁天东、傅志凌等队员。

作　者

2019 年 5 月

源码下载

目　录

第 1 章　算法竞赛概述 …… 1

1.1　培养杰出程序员的捷径 …… 2

1.1.1　编写大量代码 …… 3

1.1.2　丰富的算法知识 …… 3

1.1.3　计算思维和逻辑思维 …… 3

1.1.4　团队合作精神 …… 3

1.2　算法竞赛与创新能力的培养 …… 4

1.3　算法竞赛入门 …… 5

1.3.1　竞赛语言和训练平台 …… 5

1.3.2　判题和基本的输入与输出 …… 5

1.3.3　测试 …… 8

1.3.4　编码速度 …… 9

1.3.5　模板 …… 10

1.3.6　题目分类 …… 11

1.3.7　代码规范 …… 12

1.4　天赋与勤奋 …… 13

1.5　学习建议 …… 14

1.6　本书的特点 …… 15

第 2 章　算法复杂度 …… 17

2.1　计算的资源 …… 17

2.2　算法的定义 …… 21

2.3　算法的评估 …… 22

第 3 章　STL 和基本数据结构 …… 24

3.1　容器 …… 24

3.1.1　vector …… 25

3.1.2　栈和 stack …… 27

3.1.3　队列和 queue …… 28

3.1.4　优先队列和 priority_queue …… 30

3.1.5　链表和 list …… 30

3.1.6 set …… 31
3.1.7 map …… 33
3.2 sort() …… 34
3.3 next_permutation() …… 35

第 4 章 搜索技术 …… 37

4.1 递归和排列 …… 38
4.2 子集生成和组合问题 …… 41
4.3 BFS …… 43
4.3.1 BFS 和队列 …… 43
4.3.2 八数码问题和状态图搜索 …… 46
4.3.3 BFS 与 A* 算法 …… 50
4.3.4 双向广搜 …… 52
4.4 DFS …… 53
4.4.1 DFS 和递归 …… 53
4.4.2 回溯与剪枝 …… 54
4.4.3 迭代加深搜索 …… 57
4.4.4 IDA* …… 58
4.5 小结 …… 60

第 5 章 高级数据结构 …… 61

5.1 并查集 …… 62
5.2 二叉树 …… 66
5.2.1 二叉树的存储 …… 66
5.2.2 二叉树的遍历 …… 67
5.2.3 二叉搜索树 …… 70
5.2.4 Treap 树 …… 72
5.2.5 Splay 树 …… 78
5.3 线段树 …… 84
5.3.1 线段树的概念 …… 84
5.3.2 点修改 …… 85
5.3.3 离散化 …… 89
5.3.4 区间修改 …… 90
5.3.5 线段树习题 …… 93
5.4 树状数组 …… 93
5.5 小结 …… 97

第 6 章 基础算法思想 …… 98

6.1 贪心法 …… 98

6.1.1 基本概念 …… 98
6.1.2 常见问题 …… 100
6.1.3 Huffman 编码 …… 102
6.1.4 模拟退火 …… 105
6.1.5 习题 …… 107
6.2 分治法 …… 107
6.2.1 归并排序 …… 108
6.2.2 快速排序 …… 111
6.3 减治法 …… 113
6.4 小结 …… 114

第 7 章 动态规划 …… 115

7.1 基础 DP …… 116
7.1.1 硬币问题 …… 116
7.1.2 0/1 背包 …… 123
7.1.3 最长公共子序列 …… 127
7.1.4 最长递增子序列 …… 129
7.1.5 基础 DP 习题 …… 132
7.2 递推与记忆化搜索 …… 133
7.3 区间 DP …… 134
7.4 树形 DP …… 139
7.5 数位 DP …… 144
7.6 状态压缩 DP …… 148
7.7 小结 …… 153

第 8 章 数学 …… 154

8.1 高精度计算 …… 154
8.2 数论 …… 155
8.2.1 模运算 …… 156
8.2.2 快速幂 …… 156
8.2.3 GCD、LCM …… 159
8.2.4 扩展欧几里得算法与二元一次方程的整数解 …… 159
8.2.5 同余与逆元 …… 161
8.2.6 素数 …… 163
8.3 组合数学 …… 166
8.3.1 鸽巢原理 …… 166
8.3.2 杨辉三角和二项式系数 …… 167
8.3.3 容斥原理 …… 168
8.3.4 Fibonacci 数列 …… 168

8.3.5 母函数 …… 169
8.3.6 特殊计数 …… 174
8.4 概率和数学期望 …… 180
8.5 公平组合游戏 …… 183
8.5.1 巴什游戏与 P-position、N-position …… 184
8.5.2 尼姆游戏 …… 185
8.5.3 图游戏与 Sprague-Grundy 函数 …… 187
8.5.4 威佐夫游戏 …… 190
8.6 小结 …… 191

第 9 章 字符串 …… 192

9.1 字符串的基本操作 …… 192
9.2 字符串哈希 …… 194
9.3 字典树 …… 196
9.4 KMP …… 198
9.5 AC 自动机 …… 202
9.6 后缀树和后缀数组 …… 204
9.6.1 概念 …… 205
9.6.2 用倍增法求后缀数组 …… 206
9.6.3 用后缀数组解决经典问题 …… 212
9.7 小结 …… 213

第 10 章 图论 …… 214

10.1 图的基本概念 …… 214
10.2 图的存储 …… 215
10.3 图的遍历和连通性 …… 217
10.4 拓扑排序 …… 219
10.5 欧拉路 …… 223
10.6 无向图的连通性 …… 225
10.6.1 割点和割边 …… 225
10.6.2 双连通分量 …… 228
10.7 有向图的连通性 …… 230
10.7.1 Kosaraju 算法 …… 231
10.7.2 Tarjan 算法 …… 234
10.8 2-SAT 问题 …… 236
10.9 最短路 …… 239
10.9.1 Floyd-Warshall …… 240
10.9.2 Bellman-Ford …… 242
10.9.3 SPFA …… 246

10.9.4 Dijkstra …… 250
10.10 最小生成树 …… 253
10.10.1 prim 算法 …… 254
10.10.2 kruskal 算法 …… 255
10.11 最大流 …… 257
10.11.1 Ford-Fulkerson 方法 …… 258
10.11.2 Edmonds-Karp 算法 …… 260
10.11.3 Dinic 算法和 ISAP 算法 …… 262
10.12 最小割 …… 263
10.13 最小费用最大流 …… 264
10.14 二分图匹配 …… 268
10.15 小结 …… 271

第 11 章 计算几何 …… 272

11.1 二维几何基础 …… 272
11.1.1 点和向量 …… 273
11.1.2 点积和叉积 …… 274
11.1.3 点和线 …… 276
11.1.4 多边形 …… 280
11.1.5 凸包 …… 283
11.1.6 最近点对 …… 285
11.1.7 旋转卡壳 …… 287
11.1.8 半平面交 …… 288
11.2 圆 …… 293
11.2.1 基本计算 …… 293
11.2.2 最小圆覆盖 …… 297
11.3 三维几何 …… 300
11.3.1 三维点和向量 …… 300
11.3.2 三维点积 …… 301
11.3.3 三维叉积 …… 302
11.3.4 最小球覆盖 …… 304
11.3.5 三维凸包 …… 304
11.4 几何模板 …… 308
11.5 小结 …… 315

第 12 章 ICPC 区域赛真题 …… 316

12.1 ICPC 亚洲区域赛(中国大陆)情况 …… 316
12.2 ICPC 区域赛题目解析 …… 317
12.2.1 F 题 Friendship of Frog(hdu 5578) …… 318

12.2.2 K 题 Kingdom of Black and White(hdu 5583) …… 320
12.2.3 L 题 LCM Walk(hdu 5584) …… 323
12.2.4 A 题 An Easy Physics Problem(hdu 5572) …… 325
12.2.5 B 题 Binary Tree(hdu 5573) …… 326
12.2.6 D 题 Discover Water Tank(hdu 5575) …… 328
12.2.7 E 题 Expection of String(hdu 5576) …… 333
12.2.8 G 题 Game of Arrays(hdu 5579) …… 336
12.2.9 I 题 Infinity Point Sets(hdu 5581) …… 339

参考文献 …… 344

第 1 章　算法竞赛概述

视频讲解

- ✍算法竞赛简介
- ✍创新能力的培养
- ✍训练平台
- ✍入门知识
- ✍模板的作用
- ✍题目分类
- ✍学习计划

算法竞赛是培养大学生程序设计能力、计算思维能力、创新能力和团队合作精神的重要方式，是培养杰出程序员的捷径，被国内高校普遍重视，吸引着越来越多的大学生参与其中。

本章介绍了与竞赛相关的入门知识，包括竞赛语言及训练平台、编码规范、题目分类、模板的作用等。在本章的最后部分讨论了天赋和努力与竞赛成绩的辩证关系，并详细地给出了学习建议。

算法竞赛（程序设计竞赛）是培养杰出程序员的捷径。

在当前高等教育强化创新能力培养、逐年增大学科竞赛投入的背景下，出现了一大批面向大学生的算法类竞赛。在国内众多竞赛中，最具影响力的面向大学生的程序设计竞赛有 ACM-ICPC 和 CCPC 等，面向中学生的程序设计竞赛有全国青少年信息学奥林匹克竞赛。

ACM-ICPC（International Collegiate Programming Contest，国际大学生程序设计竞赛）[①]是国际和国内最有影响力的高校计算机竞赛，是旨在展示大学生创新能力、团队精神和在压力下编写程序、分析和解决问题能力的年度竞赛。经过多年的发展，ACM-ICPC 已经成为全球最具影响力的大学生程序设计竞赛，它也成为我国高校创新教育评估的主要竞赛之一。

CCPC（China Collegiate Programming Contest，中国大学生程序设计竞赛）[②]是由中国大学生程序设计竞赛组委会主办的面向世界大学生的国际性年度赛事，旨在激发学生学习算法和程序设计的兴趣，提升算法设计、逻辑推理、数学建模、编程实现和英语阅读能力，激励学生运用计算机编程技术和技能解决实际问题，培养团队合作意识、挑战精神和创新潜力。中国大学生程序设计竞赛是 ACM-ICPC 在中国发展的必然产物。

---

① 网址为 icpc. baylor. edu。从 2018 年起，ACM 协会不再赞助 ICPC，因此众所周知的 ACM-ICPC 竞赛应该改为 ICPC。本书由于需要介绍竞赛的历史，这里仍然沿用 ACM-ICPC 这个名称。

② 网址为 ccpc. io。

NOI(National Olympiad in Informatics,全国青少年信息学奥林匹克)[①]是国内省级代表队最高水平的大赛。每年各省选拔产生 5 名选手,由中国计算机学会组织进行比赛。

"学科竞赛不仅是高校创新人才培养的重要手段,而且是用人单位选拔人才的重要依据"[②],算法竞赛完全证明了这一点。搜索所有著名 IT 公司招聘软件工程师的面试题目就会发现,没有经过算法竞赛训练的人很难通过这些面试。

算法竞赛是展示编程能力的大舞台。练武的人,如果有一个比武的擂台,可以极大地激发他们的活力,并让他们有机会证明自己不是纸上谈兵,而是真正的高手。ACM-ICPC、CCPC、NOI 就是学习编程和展示程序设计能力的大擂台。在学习的过程中,从简单题目到难题,从一个专题到所有专题,一步一步渐进学习,让参与者能清晰地把握自己的进步。竞赛题目都是实打实的软件小项目,完成它们能给人以真实的成就感,并验证是否掌握了真正的编程本领。

1.1　培养杰出程序员的捷径

杰出的程序员有什么特质?这里可以列出一长串:掌握多种编程语言,编写过大量代码,算法知识丰富,数学应用能力强,做过很多项目,有团队精神和创新意识,善于根据行业需求调整自己的努力方向……

学习和参加算法竞赛是成为杰出程序员的捷径。ACM-ICPC 的冠军被称为"世界上最聪明的人",竞赛的获奖者基本上都成长为杰出的软件工程师,并且有很多人是 IT 公司的创业者。例如,当前最热门的人工智能公司,很多创始人都是算法竞赛的佼佼者。

依图科技联合创始人林晨曦,2002 年获 ACM-ICPC 总决赛金牌,2008 年进入阿里云任技术总监,搭建中国首个拥有自主知识产权的分布式计算平台"飞天",2012 年与同学朱珑一起联合创立依图科技。

第四范式 CEO 戴文渊,2005 年获 ACM-ICPC 总决赛金牌,在"《科学中国人》2016 年度人物"评选中,成为第一位代表人工智能行业获评"年度科技型企业家"荣誉的企业家。

旷视科技的创始人唐文斌,2008 年获 ACM-ICPC 总决赛银牌,他的公司被李开复称为"国内最成功的人工智能公司"。旷视科技公司聚集了很多算法竞赛获奖的人才。唐文斌这样评价自己公司的员工:"在我的团队里,聚集了一批这样的天才人物。目前旷视科技团队成员有 60 个左右,其中 30 多个人至少曾获得过一项世界级编程比赛奖项,得过国际奥林匹克竞赛金牌的有 7 个……"[③]

在 ACM-ICPC 区域赛上获奖的学生,在大学生中比例极小。例如,2017 年亚洲区域赛,参赛队员是从 300 个大学选拔出来的,7 个赛区合计约 1700 队。其中,金牌 10%,170 队,约 500 人,以大四学生为主;银牌 20%,340 队,约 1000 人,以大三、大四学生为主;铜牌 30%,500 队,约 1500 人,以大三学生为主。这其中有一部分队伍重复参加多个赛区的比

---

① 网址为 www.noi.cn。

② 中国高等教育学会《高校竞赛评估与管理体系研究》专家工作组。

③ 唐文斌:先打地基再建楼;http://m.iheima.com/article/150664(永久网址:perma.cc/QAE8-NKVX)。

赛，估算起来，每年毕业的金牌获奖者不到 500 人，银牌获奖者不到 1000 人。中国在校的计算机类专业大学生有 100 万左右，这还不算其他信息类专业毕业生。因此，ACM-ICPC 竞赛获奖队员可以说是千里挑一、万里挑一的杰出人才了。

在大学阶段参加算法竞赛，可以使一个未来的杰出程序员获得下面几个小节所介绍的能力。这些能力虽然被视为“基础能力”，但却是大部分学计算机编程的学生所不能轻易获得的。

1.1.1　编写大量代码

视频讲解

比尔·盖茨曾说过：“如果你想雇用一个工程师，看看他写的代码就够了。如果他没写过大量代码，就不要雇用他。”①通过编写大量代码，能做到算法精妙合理、逻辑清晰透彻、代码喷涌而出、格式赏心悦目、挑 bug 手到擒来，这是杰出程序员的基本功。ACM-ICPC 竞赛队员想达到在区域赛中获奖的水平，需要写 5～10 万行的代码。

1.1.2　丰富的算法知识

算法是程序的核心，决定了程序的优劣。特别是在数据规模大的情况下，算法直接决定了程序的生死。例如，用计算机处理排序问题：假设有 100 万个数，用最简单的冒泡排序算法，计算量可能多达 1 万亿次(冒泡排序的计算复杂度是 $O(n^2)$，100 万×100 万=1 万亿)，在计算机上，计算时间长达几个小时，实际上根本不能用；如果改用快速排序算法，计算量只有 2000 万次(快速排序的计算复杂度是 $O(n\log_2 n)$，100 万×$\log_2$100 万≈2000 万)，计算机在 1 秒内可以完成。二者的计算时间相差 5 万倍，算法的威力可见一斑。

算法竞赛涉及绝大部分常见的确定性算法，掌握这些知识，不仅能应用在软件开发中，也是进一步探索未知算法的基础。例如现在非常火爆的、代表了人类未来技术的人工智能研究，涉及许多精深的算法理论，没经过基础算法训练的人根本无法参与。

1.1.3　计算思维和逻辑思维

一些竞赛队员经常说：我们要尽量掌握所有算法知识。但是，程序设计不仅要有算法思想，还需要能正确地写出程序，这不是仅仅有算法知识就能完成的。一道难题，往往需要综合多种能力，例如数据结构、算法知识、数学方法、流程和逻辑等，这就是计算思维和逻辑思维能力的体现。通常，能解出这样的题目是高级程序员的特征。

在 ACM-ICPC 亚洲区域赛和 CCPC 赛事上获得金牌、银牌的队员，能够凭借奖牌证实自己有这样的能力。这也是算法竞赛被看重的主要原因。

1.1.4　团队合作精神

在软件行业，团队合作非常重要，这一点不需要更多说明。ACM-ICPC、CCPC 竞赛把对团队合作的要求放在了重要位置。竞赛的赛制是 3 人一队，一台计算机，十几道竞赛题，

① Gates:“If you want to hire an engineer, look at the guy's code. That's all. If he hasn't written a lot of code, don't hire him.”; https://www.wired.com/2010/04/ff_hackers/。

限定 5 个小时。参加过现场比赛的队伍都能立刻体会到：一个队伍的 3 个人，在同等水平下，如果配合默契，则可以多做一两道题，把获奖等级提高一个档次。在竞赛过程中，有人负责精读英语题，有人负责构造测试数据，有人负责编写代码，大家互相讨论思路，队长判断现场形势，确定做题顺序。每支队伍的 3 个人只有在日常训练中长期磨合，才能互相了解，做到合理分工、优势互补，从而发挥出最优的团队力量。

有人认为：毕业后参加工作，其实用不着算法竞赛这么多的复杂逻辑和算法知识，即使用到了，在工程中一般有现成的模块，拿来用就行了，只要了解这个模块用到的算法的作用和复杂度即可。这种认识是肤浅的，对于立志成为高级程序员的学生而言，进行大量的计算思维训练和经典算法训练是必需的，理由如下：

(1) 算法是对学习和理解能力的一块试金石，难的都能掌握，容易的当然不在话下。在算法竞赛上获奖的人证明了自己有解决复杂编程问题的能力。

(2) 即使有现成的模块，但是对于特定的需求，往往需要进行修改才能真正使用，没有真正理解的人无法修改。

(3) 实际的程序往往有复杂的逻辑关系，但又不属于经典的算法，没有现成模块，需要自己思考才能写出代码，这个能力是通过训练得到的。

1.2　算法竞赛与创新能力的培养

算法竞赛培养这样的能力：对复杂问题，用高效的算法或逻辑进行建模并编码实现。

目前中国的 IT 业极其繁荣，已经和美国并列为世界超级两强，把其他国家和地区远远抛到后面，并且中国有加速超过美国的趋势。繁荣意味着竞争激烈，参加编程竞赛的获奖队员能够在成千上万的 IT 工程师中脱颖而出，有很多创业并成功。本书作者认为，他们需要具备以下品质：

(1) 激情和勇气。例如，一旦开始，就不肯退缩的激情；渴望成为大人物，有改变行业的野心；敢于平等地和老师、IT 行业人士进行交流的勇气。本书作者所在的华东理工大学有一些竞赛队员毕业后创业成功，他们在大一的时候就已经表现出了这些特点。例如创办杭州美登科技有限公司(淘宝的金牌“淘拍档”)的 2008 届毕业生邹宇、创办上海萌果信息科技有限公司(中国手游企业的明星)的 2009 届毕业生尹庆，在大一的时候就表现出了不服输、敢于承担的特点，他们先后担任了竞赛队长。

(2) 开阔的思路。能抓住一切机会了解更多的信息。例如创办云片网络的华东理工大学的 2007 届毕业生刘大林，他在读大三的时候发现学校主页没有校内搜索功能，于是主动做了一个搜索，并推销给了学校。再如华东理工大学的 2012 届毕业生诸咏天，在大学期间访问了很多创业公司，开阔了思路，在大学期间就积极创业，荣获“2011 年上海市大学生年度人物”的称号，毕业后创业并取得成功。

(3) 超群的技术能力。这一点非常重要。由于现在 IT 行业早已进入成熟阶段，从业者人数太多，竞争激烈，所以只有具备超群技术能力的人才能快速开发出难以模仿的软件，增加获得成功的概率。

(4) 自信。自信不是盲目的自大，自信的获得建立在征服困难的经验上。例如，参加过

ACM-ICPC 和 CCPC 竞赛的学生，由于经历了长期的非常困难的学习，在编程能力上远远超越了大部分学生，从而建立了超强的自信。

(5) 团队建设。竞赛的队员，在长期的共同学习和参赛的过程中团结合作、共同进步，结下了深厚的友谊，是未来创业的好伙伴。

1.3 算法竞赛入门

1.3.1 竞赛语言和训练平台

ICPC 允许的算法竞赛用的编程语言有 C、C++、Java、Python、Kotlin[1] 几种。其中 C++因运行效率高、具有丰富的 STL 函数库，最受竞赛队员欢迎。Java 和 Python 也比较常用，它们在处理大数据时极为简便，这几年 Python 上升的势头很快。这几种编程语言，在就业市场上都有大量的岗位需求，极容易就业。熟练掌握一种编程语言是基本的，掌握几种语言是必要的。

竞赛队员主要的学习方法就是“刷题”，在 Online Judge(OJ，在线判题网站)上大量做编程题。OJ 上有丰富的编程题目，能对编程者提交的程序进行自动判题，返回“正确”或“错误”提示。国内、国外有很多 OJ，国内的例如 acm. hdu. edu. cn、poj. org，国外的例如 uva、ural、usaco 等[2]。OJ 的核心价值主要有两个，即题目和判题用的测试数据。测试数据的重要性不亚于题目本身，甚至更重要。

视频讲解

很多队员在高中接触了 NOI 信息学竞赛，他们常常在 CCF 的 OJ[3]，以及“洛谷”和“大视野[4]”几个网站做题，以中文题为主。其中，“洛谷试炼场[5]”的题目分类比较全，是很好的基础学习平台。

在这些 OJ 之外还有一些代理 Judge。这些代理以 http 的方式调用了宿主 OJ 提供的判题服务，连接了 30 多个著名的 OJ，相当于一个综合平台。代理 Judge 的优点如下：

(1) 方便做国外题目，因为在国内直接连接外国的 uva 等 OJ 往往极慢，而通过代理很快。

(2) 如果某个 OJ 网站直接连不上，在代理上也常常能做这些 OJ 的题目。

(3) 虚拟比赛功能，即把来自不同宿主 OJ 的题目混编为一场训练赛，特别方便日常训练[6]。

搭建 OJ 系统在技术上是不难的。事实上，几乎所有常年开展算法竞赛的学校都建立了自己的 OJ，用于训练和比赛。

1.3.2 判题和基本的输入与输出

在 OJ 提交程序后，OJ 如何判断程序是正确的还是错误的?

---

① https://icpc. baylor. edu/worldfinals/programming-environment(短网址：t. cn/Rx4xYSH)

② 参考 cn. vjudge. net 列出的 OJ 网站。

③ 全国青少年信息学奥林匹克竞赛网站：www. noi. cn；做题网站：oj. noi. cn。

④ 大视野 OJ：www. lydsy. com，网上简称 bzoj。

⑤ 洛谷试炼场：https://www. luogu. org/training/mainpage，有不错的分类。

⑥ 专题学习：kuangbin 带你飞，vjudge. net/article/187。

OJ 由计算机自动判题，但计算机并没有看懂代码的智能；即使是人工判题，人也很难在短时间内看懂程序。因此，OJ 判题是一种黑盒测试，它并不关心程序的内容，而是用测试数据来验证。

OJ 的后台存储了每个题目的测试数据，有输入数据和对应的输出数据，并且有很多组输入和输出数据。OJ 运行用户提交的程序后读取输入数据，程序产生输出，然后与后台的标准输出进行对比，可以得到以下结果之一①：

(1) 没有超时，并且完全一致，判定 Accepted (AC)。

(2) 超时，判定 Time Limit Exceeded (TLE)。在一般情况下，返回 TLE 说明方法错误，整个程序可能需要推倒重来。

(3) 结果是对的，但是格式有错误，例如多了空格，返回 Presentation Error (PE)。

(4) 结果错误，或者有其他问题，返回 WA、RE、MLE 等信息。

由于 OJ 不看程序内容，只关心程序的输入和输出，所以在程序中不写详细过程，而是用 printf()或 cout 直接打印结果，这也是允许的，这种方法叫“打表”。另外，程序可能需要预处理数据，这个做法也称为“打表”。

一个题目的测试数据可能有成千上万组。好的测试数据应该尽量覆盖所有可能的情况，而不好的测试数据会让题目失去价值，这就是为什么测试数据和题目本身一样重要的原因。

1. 输入与输出函数

C++语言中的标准输入语句为 cin，输出语句为 cout②。

C 语言中的输入与输出函数如下。

- putchar()：把一个字符常量输出到显示器屏幕上；
- getchar()：从键盘上输入一个字符常量；
- printf()：把数据按格式控制输出到显示器屏幕上；
- scanf()：从键盘上输入各类数据；
- puts()：把一个字符串常量输出到显示器屏幕上；
- gets()：从键盘上输入一个字符串常量；
- sscanf()：从一个字符串中提取各类数据。

在竞赛中，默认使用标准输入 stdin 和标准输出 stdout，所以在提交程序时并不用管 OJ 是怎么进行数据测试的。如果用到文件的输入与输出，会特别说明使用方法。

2. 输入结束方式

(1) 默认结束。在 OJ 上一般有多组测试数据，如果没有明确指出输入在什么时候结束，则程序以“文件结束”(EOF)为结束标志。例如：

```
int main(){
    int a,b;                                  //输入 a、b
```

① http://acm.hdu.edu.cn/faq.php。

② 在 *Competitive Programmer's Handbook*（作者 Antti Laaksonen，2017 年 10 月 11 日）的第 1 章中介绍了编程语言的一些注意事项。

```
    while(scanf("%d%d",&a,&b) != EOF){    //等价于 while(~scanf("%d%d", &a, &b)){
      …;
    }
    return 0;
}
```

一些队员喜欢把输入语句写成：

```
while(~scanf("%d%d", &a, &b))
```

这也是对的。因为如果没有输入，scanf()返回 EOF，系统定义 EOF = −1，取非就是 0。

在竞赛时，一般不建议用判断 EOF 的方法。本书的程序采用 while (~scanf("%d%d", &a, &b))形式。

(2) 在输入数据中指定了数据个数。一般在输入数据的第 1 行定义数据量大小，例如第 1 行是 100，则表示有 100 组数据。这里以 hdu 1090 题为例，程序如下：

hdu 1090 题程序

```
int main(){
    int n, a, b;
    scanf("%d", &n);                     //n: 有多少组数据
    while(n--){
        scanf("%d %d", &a, &b);
        printf("%d\n", a + b);
    }
    return 0;
}
```

(3) 以特定元素作为结束符。例如以 0 作为结束符，当输入读到 0 时就退出，可以这样写：

```
while(~scanf("%d",&n) && n)
```

3. 输入与输出的效率

在 C++语言中，输入和输出常用的语句是 cin、cout，优点是很方便。但是用户需要注意，与 scanf()、printf()相比，cin、cout 的效率很低，速度很慢。如果题目中有大量的测试数据，由于 cin、cout 输入和输出慢，可能导致 TLE，在这种情况下应使用 scanf()、printf()。

例如 hdu 3233 题。在本例中输入 $1 \leqslant T \leqslant 20\,000$，可能有 20 000 行数据，因此输入的效率很关键。此题用 scanf()、printf()可以 AC，OJ 返回的执行时间是 140ms；用 cin、cout，结果 TLE，执行时间超过 1000ms。

hdu 3233 程序：用 scanf()、printf()，AC，执行时间是 140ms

```
#include<bits/stdc++.h>
int main(){
    int T, n, cnt = 1, B;
    while(scanf("%d%d%d", &T, &n, &B)){
```

```
        if(T == 0 || n == 0 || B == 0) break;
        double s, p, sum = 0;
        while(T -- ) {                          //1≤T≤20000
            scanf("%lf%lf", &s, &p);            //高效率输入
            sum += s * (100 - p) * 0.01;
        }
        printf("Case %d: %.2f\n\n", cnt++, sum/(B * 1.0));
    }
    return 0;
}
```

hdu 3233 程序：用 cin 和 cout，结果 TLE，执行时间超过 1000ms

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int T, n, cnt = 1, B;
    while(cin >> T >> n >> B){
        if(T == 0 || n == 0 || B == 0) break;
        double s, p, sum = 0;
        while(T -- ) {                          //1≤T≤20000
            cin >> s >> p;                      //输入很慢
            sum += s * (100 - p) * 0.01;
        }
        cout << "Case " << cnt++ << ": " << fixed << setprecision(2)
            << sum/(B * 1.0) << endl << endl;
    }
    return 0;
}
```

1.3.3 测试

1. 构造测试数据

在程序编好之后应该自己先测试通过，再提交到系统，而题目给的样例数据一般都太少，不足以检验程序的正确性，队员需要自己构造测试数据。

在一个队伍中，一般安排一个队员专门负责构造测试数据。对于需要高级算法的题目，可以让这名队员先用暴力法编程，然后随机生成输入数据，运行暴力程序，生成输出数据。输入数据除了随机生成以外，有时候还需要手工生成一些，主要是边界数据、特别小的数据、特别大的数据等，这些也是最容易出错的。

为方便操作，可以把构造出的输入数据放在文件 test.in 里，将程序的结果输出到文件 test.out 里。当然，有时候不需要输出文件 test.out，直接在屏幕上看输出结果就可以了。

那么如何方便地使用它们？有以下两种方法：

(1) 在程序中加入测试代码。

```
#define mytest
#ifdef mytest
```

```
    freopen("test.in", "r", stdin);
    freopen("test.out", "w", stdout);
#endif
```

在提交时，去掉 #define mytest 即可。

(2) 在行命令中重定向。

这是更简单的方法，不用在程序中加任何代码。例如，生成的可执行程序是 abc，在 Windows 或 Linux 的行命令中这样输入和输出到文件：

```
abc < test.in > test.out
```

2. 对比测试数据

视频讲解

对于复杂的题目，可能需要写两个程序：一个是提交到 OJ 的"好"程序；另一个是暴力法程序，目的是用它生成测试数据。这种方法称为"对拍"。

在测试的时候，可以用行命令比较两个程序的输出是否一致。例如在 Windows 系统下，生成的可执行文件分别是 abc.exe、abc_1.exe，用文件比较命令 fc 比较它们的输出是否一致。

```
abc.exe < test.in > test1.out
abc_1.exe < test.in > test2.out
fc test1.out test2.out /n
```

在 Linux 系统中，文件比较命令是 diff。

1.3.4　编码速度

竞赛时间很紧张，编码应该简洁。跟软件工程的代码相比，竞赛题的代码都不长，从几行到 200 多行。快速编程得到正确结果即可，无须担心程序是否符合工程项目的要求，也不要求写得多么"漂亮"，这是竞赛中编码的特点。

编程速度决定了参赛获奖的级别。在一般情况下，同样的出题数量会跨越相邻的获奖等级，例如同样做 5 道题，排前面的获银牌，排后面的获铜牌。但是，如果跨越的等级过大，则是不正常的。近些年来，由于区域赛的出题质量参差不齐，"速度"这个因素的影响越来越大。一套理想的题目应该有很好的区分度，例如金牌 8 题以上，银牌 6 题以上，铜牌 4 题以上。但是近年来常常遇到这样的赛区：终榜时，做题数量一样，只是因为出题快慢不同，名次就从银牌到铜牌再到铁牌(铁牌是 honorable mention，即鼓励奖的玩笑说法)，分成了 3 个等级。应该说，这样大的跨越度不能区分参赛队伍的水平。

那么如何提高编码速度？

(1) 读题要快。题目都是英文的，新队员往往不习惯，需要长期训练才能适应。虽然有些小窍门，例如先读样例，再读题面内容，但最根本的还是靠大量的英语阅读练习，学会在脑海中直接用英语进行思考，才能提高读题速度。一套题需要 3 个队员分工快速读完，每个人读完后必须和队员一起讨论，确定完全理解题意。在竞赛现场紧张的气氛下，一个队员不要太相信自己，而忽视了队友的帮助。

(2) 熟练掌握编辑器或 IDE。根据规定，现场赛提供的编辑器有 vim、gedit 等，IDE 有

Eclipse、Code::Blocks 等[1]。Eclipse 和 Code::Blocks 受到新手和很多老队员的欢迎，不过一些老队员说，熟练使用编辑器 vim 写代码速度更快。在竞赛中，能赢得几分钟的领先时间有时是很关键的。据说，参加世界总决赛的队员使用 vim 的比例很高。

(3) 不要“霸占”计算机。由于竞赛时是三人一机，只能有一人使用键盘输入代码，另外两人在旁边手写代码，等计算机空闲了再使用。在平时训练时应该养成事先在纸上写好代码的习惯，不能边敲键盘边思考，否则会浪费机时。

(4) 减少调试。因为机时非常宝贵，所以除必要的代码输入和测试外尽量少使用计算机。在写好程序后，争取能一次通过测试样例。

为了减少调试，尽量使用不容易出错的方法，例如少用指针、使用静态数组、把逻辑功能模块化等。

另外，不要使用动态调试方法，不要用单步跟踪、断点等调试工具。如果需要查看中间数据，可用 cout()或 printf()打印出调试信息。

如果程序有问题，不要在计算机上检查，应该打印出来坐在旁边看，把机时让给队友。

(5) 互相检查。把代码讲给队友听是查错的好办法，即使队友不能理解你的思路，你在讲解的过程中也往往能突然发现自己的问题。

(6) 使用 STL。如果题目涉及比较复杂的数据处理，或者像 sort()这样需要灵活排序的函数，用 STL 可以大大减少编码量，并减少错误的发生。例如第 10 章的最短路径算法 Dijkstra，需要对结点进行松弛处理，自己编程实现会很烦琐，而如果直接使用 STL 的优先队列，编码能极大简化。

(7) 一些编码小技巧。例如把长字符重新定义成短字符，可以节省一点时间：

```
typedef  long  long  ll;
```

那么

```
long long a = 1234567890;
```

变成了简洁的：

```
ll a = 1234567890;
```

1.3.5 模板

使用模板对提高编码速度很有帮助。

刚参加算法竞赛学习的队员都听说有一种叫“模板”的神器。模板是参赛选手认为有用的代码片段，将其打印出来，允许带进竞赛现场作为“小抄”。在网上能找到很多老队员的模板，它们是学习编码的好的参考。

听起来“模板”是非常有用的：竞赛涉及几百种数据结构和算法知识，如果把它们的经典代码都总结出来，在做题的时候直接拿来用不就行了吗？这不就是软件工程的“模块化”吗？

---

① https://icpc.baylor.edu/worldfinals/programming-environment，规定了编程环境（短网址：t.cn/Rx4xYSH）。

现实也证明了这一点：有些赛区确实会出一些“模板题”，模板上的程序模块真的能直接应用在竞赛题中。

模板非常有用，其重要性主要在于帮助参赛选手理解经典算法，而不一定能用在赛场上。

使用模板需要考虑以下问题：

(1) 模板题并不常见。一个负责任的现场赛会避免出模板题，所有的题目都不能直接抄模板。

(2) 抄模板的能力。即使有模板，就一定会抄吗？模板的代码需要自己真正理解，并多次使用过，这样才能在做题的时候快速应用到编码中。不同的编程题目，即使用到相同的算法或数据结构，也往往不能用同样的代码，而需要做很多修改，因为不同环境下的变量和数据规模是不同的。因此，对模板的学习和使用需要花时间融会贯通，不能急躁。期望靠模板速成、急于拿去参赛获奖是没用的。

(3) 综合模板的能力。即使能用到模板，但是题目往往需要综合几个算法、逻辑、数据结构等，如何把模板融入整体代码中是很考验参赛选手能力的。如果只会用模板而没有理解模板，就像把尺寸不匹配的齿轮摁在一起，根本转不起来。

(4) 最重要的一点是**建模能力**。一个好题目符合这样的特征：题目很清晰，问题很清楚，然而很难想出它是什么算法、能用什么模板。但是，如果有人直接告诉你它其实是什么算法，可以用什么模板，你会恍然大悟，很快就能做出来。这个能力就是建模能力。真正的学习是掌握算法后面的思想，而不是只会背算法。通常有这样的比喻：若只会使用算法的模板而没有深入掌握算法的思想，则这个模板相当于残疾人的假肢，看起来像是自己的，走起来就知道不是自己的。只有把算法的思想深深根植于脑海，才会使其成为身体的一部分，达到心手一致的境界。从这个角度出发也能理解“质＞量”，在学习时不要追求学到的算法的“数量”，而是要掌握其“思想”。很多算法的思想其实是相通的。

本书所讲解的程序代码都经过了精心准备，是经典代码，大部分可以当模板学习。

每个队员都需要总结自己的模板。在比赛时带到赛场上，也许真的会遇到模板题呢！

提高编程速度，最根本的还是要通过大量练习，提高编码的熟练程度，“无他，但手熟尔！”

1.3.6 题目分类

算法竞赛涉及很多方面的知识，可以粗略地进行以下分类①：

- Ad Hoc，杂题；
- Complete Search (Iterative/Recursive)，穷举搜索(迭代/回溯)；
- Divide and Conquer，分治法；
- Greedy (usually the original ones)，贪心法；
- Dynamic Programming (usually the original ones)，动态规划；
- Graph，图论；
- Mathematics，数学；

① 在《Competitive Programming 3》(作者 Steven Halim、Felix Halim)的“1.2.2 Quickly Identify Problem Types”中。另外，在“1.4 The Ad Hoc Problems”中列出了大量杂题，读者可以做一做。

- String Processing，字符串处理；
- Computational Geometry，计算几何；
- Some Harder/Rare Problems，罕见问题。

除杂题外，其他分类都与数据结构和算法有关。

杂题也很常见，每次现场赛都会有 1 道或 2 道题。虽然程序设计竞赛的重点在算法方面，但是竞赛时有些题只考查逻辑能力和编码能力，并不涉及数据结构或算法，只要学过基本 C++语法就能做。这样的题目可能很难。作为练习，读者可以尝试下面的题目，它们都是大型模拟题，以烦琐、坑人著称，代码超过 200 行，需要很大的耐心和细心：

- bzoj 1972[1]"猪国杀"；
- bzoj 1033"杀蚂蚁"；
- bzoj 2548"灭鼠行动"。

本书内容包括杂题以外的所有分类，在每个分类中均会讲解常用的知识点。

1.3.7 代码规范

虽然说每个程序员都可以有自己的编码风格，但是还需要遵循大家公认的一些规范，以便于和队友互相交流。

下面列出了一些常见的注意事项。

(1) header。使用万能头文件"#include <bits/stdc++.h>"，OJ 网站一般都支持，一个例外是 poj，它不支持。

另外，不要用 C 风格的 header，例如#include <stdio.h>。

(2) 输入判断结尾不要用 EOF，而用 '~'，例如：

```
~scanf("%d", &n)
```

在 1.3.2 节中已经详细介绍了这个问题。

(3) 换行。用 K&R 风格，即左大括号不换行，右大括号单列一行。

(4) 变量定义。变量定义在这个变量被调用的最近的地方，例如：

```
for (int i = 0; i < 10; i++) {               //i 只在这个循环体内使用
    int s = i * i;
}
```

(5) 最好不要用宏。不管是宏定义还是宏函数，都容易出问题。

不要用#define 定义常量，而用 const 定义常量，例如：

```
const int MAX = 1000005;
```

视频讲解

把宏函数写成普通函数。

(6) 参考资料。

Google C++规范：https://google.github.io/styleguide/cppguide.html。

Linux C 规范：https://www.kernel.org/doc/html/v4.10/process/coding-style.html。

---

[1] https://www.lydsy.com/JudgeOnline/problem.php?id=1972(短网址：t.cn/RgCa1Si)。

1.4 天赋与勤奋

世上是否存在编程天才？答案是肯定的。普通智力能否达到很高的编程水平？答案也是肯定的。人们常说的“天赋决定上限，努力决定下限”，在编程竞赛这种高智力活动上有一定的道理。

天赋，在成功的因素中占有很大的比重。如果要达到顶级水平，天赋就更重要了。例如在体育运动项目中，能达到获得奥运奖牌水平的运动员，他的天赋几乎有决定性的影响。在脑力活动中，类似编程这样繁杂而高深的思维活动，智力的因素非常大。

如果读者有兴趣，可以用五子棋或魔方检验自己在记忆力、逻辑推理、空间想象力、专注度、敏捷性等方面的智力天赋。这两种游戏的特点是规则简单、上手快、变化比较复杂。所有人都能玩，但是想玩好，大部分人需要一段比较长的学习时间。一个初学者，如果只需要几天的学习就能达到很高的水平，那么差不多可以说他拥有编程天赋了[①②]。

这些有编程天赋的少数人，如果能专注练习，他们的学习效率要比普通人高出几倍，更容易成功。如果他们在刚上大学的时候从零基础开始学习编程，那么他们在大三甚至大二就能获得银牌，在大四或者大三就能获得金牌，可称为天之骄子！

智力普通的学生，通过勤奋的学习，挖掘出自己的智商潜力、锻炼自己的专业技能，也能达到很高的水平。特别是对于编程这种需要掌握海量知识、拥有长期编码经验的高智力活动来说，勤奋相对天赋的比重在职业生涯中会越来越大。根据经验，参加 ICPC 竞赛的学生，即使是零起点，如果能在大一入学后坚持每天 2～4 个小时的编程学习，那么他完全可以在大三的第一学期参加区域赛并获得铜牌，甚至银牌、金牌。

学习编程需要做好艰苦学习的心理准备。编程是一个长期、艰苦的过程，有乐趣，更有挫折。

在标题为“Why Learning to Code is So Damn Hard”的网页中[③]对学习编程的不同阶段给出了一个有趣的图，如图 1.1 所示。

该图把编程分成 4 个阶段，横坐标是编码能力，纵坐标是信心。

第 1 阶段(hand-holding honeymoon)：手把手关怀的蜜月期，能力和信心同步增长。初学者充满了乐趣，很有成就感，能找到丰富的学习资料。

第 2 阶段(cliff of confusion)：充满迷惑的下滑期。虽然编程者的实际能力在上升，但却逐渐丧失了信心。这是因为遇到了难以解决的问题、需要调试大量 bug、遇到挫败。不过这个时候仍能够找到答案，知识面也在变广。

第 3 阶段(desert of despair)：绝望的迷茫期，信心的沙漠。编程者遇到更加困难的问题，需要的知识剧增，但是资源匮乏，在网上也找不到答案，或者不知道该怎么提问，感觉就

① 一节课领悟五子棋：www.zhihu.com/question/265407029/answer/299115371(永久网址：perma.cc/uz27-BXKT)。

② 天才女程序员：www.zhihu.com/question/29784784(永久网址：perma.cc/XS4R-45ZS)。

③ www.vikingcodeschool.com/posts/why-learning-to-code-is-so-damn-hard(永久网址：perma.cc/BK4R-WS7F)这条曲线和 Dunning-kruger effect 的曲线很相似，与“认知偏差”有关。

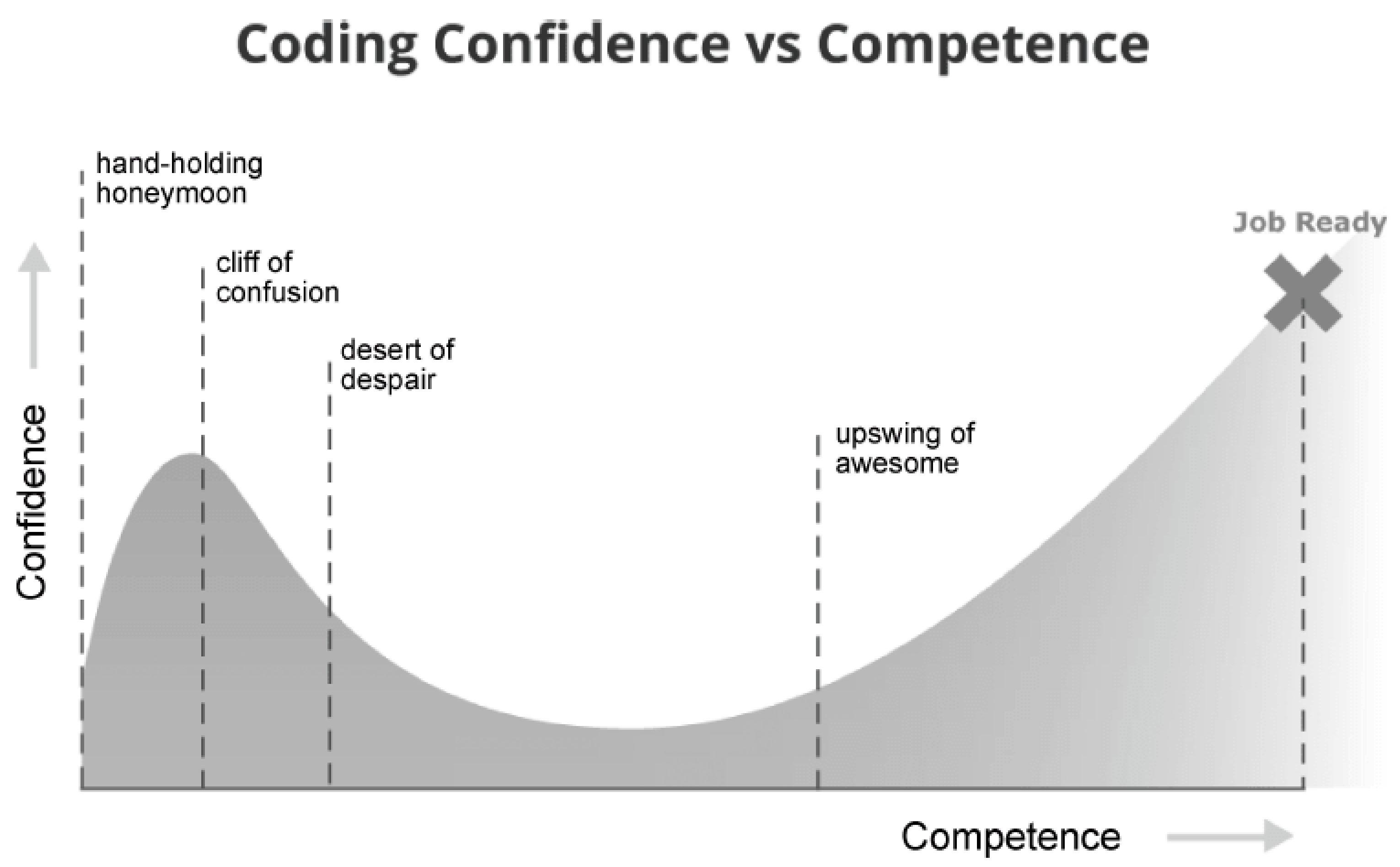


图 1.1 编码能力与信心的关系

像在沙漠一样。

第 4 阶段(upswing of awesome)：煎熬的上升期。编程者心潮澎湃，浑身充满力量，绝望的沙漠已经过去。

简单地说，ICPC 区域赛铜牌水平的选手可能还未到达第 4 阶段；银牌以上水平的选手，可以确定到达了第 4 阶段，从而跨过了自由编程的门槛。

本书的内容，在这个图中估计只涉及整个阶段的前 30%，即前两个部分，也就是 hand-holding honeymoon 和 cliff of confusion，相当于入门和进阶。能否进入后两个阶段，取决于读者自己的努力。

1.5 学习建议

很多大学生在中学阶段就参加过 NOI 信息学竞赛，或者学习过编程，那么他们已经有了基础，进入大学后又投入了更多时间专心地进行编程训练，有这么好的起点当然是很有优势的。

如果是完全的零基础，也不用担心自己落后。因为相比已经有了基础的同学只是晚学了几个月而已，只要多花一些时间，很快就能赶上。对于算法竞赛这样需要两三年的长周期学习来说，坚持才是最重要的。

由于算法竞赛的艰难和长期性，不管有没有基础，都应该从大一上学期开始学习。

(1) 大一上学期，熟悉 C、C++、Java 语言。一些专业在大一上学期开设编程语言课；大部分专业是在大一下学期，这些学生需要自学编程语言。

(2) 大一上学期，做一些简单的中文题，例如 acm. hdu. edu. cn 的 2000～2099 题[①]、洛谷试炼场。任务是进一步熟悉编程语言、学习如何在 OJ 上做题、掌握输入与输出的用法、

① http://acm. hdu. edu. cn/listproblem. php?vol=11，大部分比较简单，也有难题，可以跳过。

积累代码量。基本上每个题目在网上都能搜到题解和代码。初学者可以多看看别人的代码，尽快提高自己的编码能力。另外，最好几个人一起编程，并互相改错。看懂别人的代码，找出别人代码的错误，也是很好的训练，重要性**不亚于**独立做题。

(3) 大一下学期，做一些入门题，例如搜索、数学、贪心、简单动态规划等，尽可能多地参加各校举办的新生网络赛。

(4) 大一暑假，参加集训，学习数据结构、深入掌握STL、进行各种专题入门，并熟悉队友。

(5) 大二，深入各类专题学习，并制定一年的计划，牢固掌握各种算法知识点。如有可能，在大二上学期参加区域赛。

(6) 大二暑假，组队参加网络赛和模拟赛。

(7) 大三上学期，参加区域赛并获奖。

(8) 大三和大四，开始难题、综合题的学习，使自己获得彻底的飞越，成为"编码大师"。通常，能获得金牌的队伍至少能做出1个以上的难题。难题有3个特征，即综合性强、思维复杂、代码冗长。这些难题是绝大部分学编程的学生难以翻越的大山，能征服大山的竞赛队员可以称为"杰出"了。

1.6　本书的特点

参加竞赛训练的队员需要阅读各种各样的算法书、编程书。本书作者不奢望写出一本面面俱到、老少咸宜的书，而且这样的书也许并不存在。本书面向的读者是初学者和中级进阶者，特点如下：

(1) 算法知识点的讲解清晰、易懂。在众多确定的算法中，少数算法比较简单，多数比较难。通俗易懂地讲解一个复杂的算法是不容易的。对于初学者，经常发生的场景是在学习某个知识点的时候读了很多书，查阅了很多资料，却仍然头脑昏昏，不得要领，在痛苦地花了很多时间思索之后才恍然大悟。本书的编写目标是让读者"一点就透，豁然开朗"，因此，书中大量使用了比喻、图解、步骤、注解等方法，尽量降低初学者的学习难度。

(2) 例题简单、直接。为了讲清楚算法，每个算法都需要配合竞赛题和代码，了解应用模型，并把理论和编码结合起来。本书选择的例题大多是简单、直接的"裸题"，很少有综合题、转弯题。也就是说，本书的目的是"奠基"，以及构建算法知识门类的"框架"，上面的高楼需要读者自己去建设。对于难题、综合题，已经有很多题解被编成书出版，读者也可以到网上搜索题解，网上的资源更加丰富。

(3) 代码清晰、易读。准确、清晰的代码能让读者对算法知识的理解更加透彻。本书的每段代码都以成为"模板"为目标。这些代码是在借鉴大量代码的基础上提炼出来的，是作者精心总结的结晶。

(4) 尽量覆盖竞赛的知识体系。虽然本书只能讲解部分常用的知识点，但是每一章都对相关知识点做了介绍，希望初学者在阅读本书的同时扩展本书未能讲解的知识。

总之，**本书不是一本题解**，而是一本**帮助读者建立计算思维**的书，旨在帮助读者打造坚实的基础，获得继续深入的信心。

最后，用下面的讨论作为本章的结语。

算法竞赛涉及的知识非常多，有些算法在竞赛中常用，有些不常用。如果初学者过于功利，就会纠结于哪些算法该学，哪些不该学。

作者的看法是，学习算法并不只是为了参加竞赛。每个经典算法都经过了无数人的精心研究，是极为精巧、合理的思维运动，是计算机科学这片天空的星星。学习它们，本身就是很好的思维练习，比做多少竞赛“热题”都强得多！

第2章　算法复杂度

✍计算的资源
✍算法的定义
✍算法的评估

编程初学者肯定思考过这样一个问题：拿到一个计算问题，自己尝试编程解决，衡量这个程序好坏的标准是什么？

结果正确、运行速度快、程序结构优美、算法设计合理等，这些都可以成为衡量的标准。不过，选择用什么算法是最根本的问题，它决定了程序能用还是不能用。

本章将回答以下问题：什么是算法？为什么要使用该算法？如何评价算法的好坏？如何选择算法？通过对这些问题的讲解帮助算法竞赛的初学者建立基本的计算思维。

2.1　计算的资源

程序运行时需要的资源有两种，即计算时间和存储空间。资源是有限的，一个算法对这两个资源的使用程度可以用来衡量该算法的优劣。

- 时间复杂度：程序运行需要的时间。
- 空间复杂度：程序运行需要的存储空间。

与这两个复杂度对应，OJ上的题目一般会有对运行时间和空间的说明，例如：

Time Limit：2000/1000ms(Java/Others)

Memory Limit：65 536/65 536KB(Java/Others)

Time Limit是对程序运行时间的限制，这个题目要求在2s(Java)/1s(C、C++)内结束。

Memory Limit是对程序使用内存的限制，这里是65 536KB，即64MB。

这两个限制条件非常重要，是检验程序性能的参数。不过在现场赛中，为了增加迷惑性，可能**不会列出**这两个参数，需要参赛队员自己判断。

所以，队员拿到题目后，第一步要分析的是程序运行需要的**计算时间**和**存储空间**。

编程竞赛的题目，在逻辑、数学、算法上有不同的难度：简单的，可以一眼看懂；复杂的，往往需要很多步骤才能找到解决方案。它们对程序性能考核的要求是程序必须在限定的时间和空间内运行结束。

这是因为，问题的“有效”解决不仅在于能否得到正确答案，更重要的是能在合理的时间和空间内给出答案。

李开复在“算法的力量”一文中写道：“1988年，贝尔实验室副总裁亲自来访问我的学校，目的就是想了解为什么他们的语音识别系统比我开发的慢几十倍，而且，在扩大至大词汇系统后速度差异更有几百倍之多……在与他们探讨的过程中，我惊讶地发现一个$O(nm)$

的动态规划居然被他们做成了 $O(n^2m)$……贝尔实验室的研究员当然绝顶聪明，但他们全都是学数学、物理或电机出身，从未学过计算机科学或算法，这才犯了这么基本的错误。”

上文提到的 $O(nm)$ 和 $O(n^2m)$ 就是时间复杂度。符号 O 表示复杂度，$O(nm)$ 可以粗略地理解为运行次数是 $n\times m$。$O(n^2m)$ 比 $O(nm)$ 运行时间差不多大 n 倍。

在上面这个语音识别的例子中，设 $n=100$，如果李开复的识别系统的运行时间是 1s，那么贝尔实验室的系统需要 100s。显然，一个长达 100s 才能得到结果的语音识别系统肯定是不实用的。李开复的这个例子生动地说明了“好”算法的属性——有合理的时间效率。

那么如何衡量程序运行的时间效率？测量程序在计算机上运行的时间，可以得到一个直观的认识。

下面的程序只有一个 for 语句，它对 k 进行累加，循环次数是 n。该程序用 clock() 函数统计 for 循环的运行时间。

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int i, k, n = 1e8;
    clock_t start, end;
    start = clock();
    for(i = 0; i<n; i++)  k++;            //循环次数
    end = clock();
    cout << (double)(end - start) / CLOCKS_PER_SEC << endl;
}
```

上面的程序在一台普通配置的计算机上运行，例如 CPU 为 i5-8250U、内存为 8GB、64 位的操作系统，结果如下：

当 $n=1e8=10^8$ 时，输出的运行时间是 0.164s。

当 $n=1e9$ 时，输出的运行时间是 1.645s。

评测用的 OJ 服务器，性能可能比这个好一些，也可能差不多。

所以，如果题目要求“Time Limit：2000/1000ms(Java/Others)”，那么内部的循环次数应该满足 $n\leqslant 10^8$，即 1 亿次以内。

由于程序的运行时间依赖于计算机的性能，不同的计算机结果不同，所以直接把运行时间作为判断标准并不准确。通常，用程序执行的“次数”来衡量更加合理，例如上述程序循环了 n 次，把它的运行效率记为 $O(n)$。

竞赛所给的题目一般都会有多种解法，它考核的是在限定时间和空间内解决问题。如果条件很宽松，那么可以在多种解法中选一个容易编程的算法；如果给定的条件很苛刻，那么能选用的合适算法就不多了。

下面用一个例子来说明对同样的问题如何选用不同的解法。

hdu 1425 “sort”

Time Limit：6000/1000ms(Java/Others)Memory Limit：64/32MB(Java/Others)

给出 n 个整数，请按从大到小的顺序输出其中前 m 大的数。

输入：每组测试数据有两行，第 1 行有两个数 n 和 m（$0<n, m<1\,000\,000$），第 2 行包含 n 个各不相同，且都处于区间[−500 000, 500 000]的整数。

输出：对每组测试数据按从大到小的顺序输出前 m 大的数。

输入样例：

5 3

3 −35 92 213 −644

输出样例：

213 92 3

该题的思路是先对 100 万个数排序，然后输出前 m 大的数。题目给出了代码运行时间，非 Java 语言的时间是 1s，内存是 32MB。

下面分别用冒泡排序、快速排序、哈希 3 种算法编程。

1. 冒泡排序

首先用最简单的冒泡排序算法求解上面的问题。

```
#include <bits/stdc++.h>
using namespace std;
int a[1000001];                                       //记录数字
#define swap(a, b) {int temp = a; a = b; b = temp;}  //交换
int n, m;
void bubble_sort(){                                   //冒泡排序,结果仍放在 a[]中
    for(int i = 1; i <= n - 1; i++)
        for(int j = 1; j <= n - i; j++)
            if(a[j] > a[j + 1])
                swap(a[j], a[j + 1]);
}
int main(){
    while(~scanf("%d%d", &n, &m)){
        for(int i = 1; i <= n; i++)  scanf("%d", &a[i]);
        bubble_sort();
        for(int i = n; i >= n - m + 1; i--){          //打印前 m 大的数,反序打印
            if(i == n - m + 1)  printf("%d\n", a[i]);
            else printf("%d ", a[i]);
        }
    }
    return 0;
}
```

在 bubble_sort()运行后，得到从小到大的排序结果，然后从后往前打印前 m 大的数。冒泡排序算法的步骤如下：

(1) 第一轮，从第 1 个数到第 n 个数，逐个对比每两个相邻的数 a、b，如果 $a>b$，则交换。这一轮的结果是把最大的数“冒泡”到了第 n 个位置，在后面不用再管它。

(2) 第二轮，从第 1 个数到第 $n-1$ 个数，对比每两个相邻的数。这一轮，把第二大的数

“冒泡”到了第 $n-1$ 个位置。

(3) 继续以上过程,直到结束。

下面分析程序的时间和空间效率。

(1) 时间复杂度,也就是程序执行了多少步骤,花了多少时间。

在 bubble_sort()中有两层循环,循环次数是 $n-1+n-2+\cdots+1\approx n^2/2$;在 swap($a$,$b$)中做了 3 次操作;总的计算次数是 $3n^2/2$,复杂度记为 $O(n^2)$。当 $n=100$ 万时,计算超过 1 万亿次。如果提交到 OJ,由于 OJ 每秒只能运行 1 亿次,必然返回 TLE 超时。可以推出,只有 $n<1$ 万时才勉强能用冒泡算法。

(2) 空间复杂度,也就是程序占用的内存空间。程序使用 int a[1000001]存储数据,bubble_sort()也没有使用额外的空间。int 是 32 位整数,占用 4 个字节,所以 int a[1000001]共使用了 4MB 空间。这是冒泡算法的优点,它不额外占用空间。

2. 快速排序

快速排序是一种基于分治法的优秀排序算法。这里先直接用 STL 的 sort()函数,它是改良版的快速排序,称为“内省式排序”。

在上面的程序中,把“bubble_sort();”改为“sort(a+1, a+n+1);”就完成了 a[1]到 a[n]的排序,结果仍然保存在 a[]中。

算法的时间复杂度是 $O(n\log_2 n)$,当 $n=100$ 万时,100 万×$\log_2$100 万≈2000 万。

在 hdu 上提交,返回的运行时间是 600ms①,正好通过 OJ 的测试。

3. 哈希算法

哈希算法是一种以空间换取时间的算法。本题的哈希思路是:在输入数字 t 的时候,在a[500000 + t]这个位置记录 a[500000 + t] = 1,在输出的时候逐个检查 a[i],如果 a[i]等于 1,表示这个数存在,打印出前 m 个数。程序如下:

```
#include<bits/stdc++.h>
using namespace std;
const int MAXN = 1000001;
int a[MAXN];
int main(){
    int n,m;
    while(~scanf("%d%d", &n, &m)){
        memset(a, 0, sizeof(a));
        for(int i=0; i<n; i++){
```

① 此题数据量很大,大量时间花在输入上,如果用 cin 输入,会 TLE,真正花在排序上的时间不多。读者可能有兴趣了解具体的执行时间,可以非常粗略地分析如下:

(a) 快排程序用了 600ms,包括输入与输出时间 A,以及排序时间 B。

(b) 哈希程序用了 500ms,它的输入与输出时间和快排程序的时间差不多,都是 A;排序时间是 C。

(c) 快排的复杂度是 $O(n\log_2 n)$,哈希的复杂度是 $O(n)$。当 $n=100$ 万时,$\log_2$100 万≈20,那么 B≈20C。计算得到 A≈500ms,B≈100ms,C≈5ms。也就是说,程序的大部分时间花在了输入与输出上。

(d) 分析 $n=200$ 万的情况。快排程序的总时间≈2A+B×(200 万×$\log_2$200 万)/(100 万×$\log_2$100 万)≈2A+2.1B=1210ms;哈希程序的总时间≈2A+2C=1010ms。看起来似乎改善不大,因为时间大部分用在处理输入与输出上。如果一个程序的输入用时不多,那么时间就取决于排序算法。哈希算法比快排算法快 $\log_2 n$ 倍,很有优势。

```
            int t;
            scanf("%d", &t);            //此题数据多,如果用很慢的 cin 输入,肯定 TLE
            a[500000 + t] = 1;          //数字 t,登记在 500000 + t 这个位置
        }
        for(int i = MAXN; m > 0; i -- )
            if(a[i]){
                if(m > 1)  printf("%d ", i - 500000);
                else       printf("%d\n", i - 500000);
                m -- ;
            }
    }
    return 0;
}
```

程序并没有做显式的排序，只是每次把输入的数按哈希插入到对应位置，只有 1 次操作；n 个数输入完毕，就相当于排好序了。总的时间复杂度是 $O(n)$。在 hdu 上提交，返回的运行时间是 500ms。

4. 算法的选择

从上述 3 种程序可知，对于同一个问题，经常存在不同的解决方案，有高效的，也有低效的。算法编程竞赛主要的考核点就是在限定的时间和空间内解决问题。虽然在大部分情况下只有高效的算法才能通过满足判题系统的要求，但是请注意，并不是只有高效的算法才是合理的，低效的算法有时也是有用的。对于程序设计竞赛来说，由于竞赛时间极为紧张，解题速度极为关键，只有尽快完成更多的题目才能获得胜利。在满足限定条件的前提下，用最短的时间完成编码任务才是最重要的。低效算法的编码时间往往大大低于高效算法。例如，题目限定时间是 1s，现在有两个方案：①高效算法 0.01s 运行结束，但是代码有 50 行，编程 40 分钟；②低效算法 1s 运行结束，但是代码只有 20 行，编程 10 分钟。显然，此时应该选择低效算法。

不过在竞赛时，这种情况通常只发生在数据规模小的简单题中，即所谓的“签到题”，而大部分题目是没有这种好事的。所以，这只是一个小小的技巧，并没有太大用处。

2.2　算法的定义

前面反复提到了“算法”这个概念，参加 ACM-ICPC、CCPC 竞赛的学生也常常说“我们在搞算法竞赛”。大家也常常听说“程序＝算法＋数据结构”，算法是解决问题的逻辑、方法、过程，数据结构是数据在计算机中的存储和访问方式，二者是紧密结合的。

算法(Algorithm)是对特定问题求解步骤的一种描述，是指令的有限序列。它有以下 5 个特征。

(1) 输入：一个算法有零个或多个输入。程序可以没有输入，例如一个定时闹钟程序，它不需要输入，但是能够每隔一段时间就输出一个报警。

(2) 输出：一个算法有一个或多个输出。程序可以没有输入，但是一定要有输出。

(3) 有穷性：一个算法必须在执行有穷步之后结束，且每一步都在有穷时间内完成。

(4) 确定性：算法中的每一条指令必须有确切的含义，对于相同的输入只能得到相同的输出。

(5) 可行性：算法描述的操作可以通过已经实现的基本操作执行有限次来实现。

这里以冒泡排序算法为例，上一节已经描述过它的执行步骤，它满足上述 5 个特征。

(1) 输入：由 n 个数构成的序列$\{a_1, a_2, a_3, \cdots, a_n\}$。

(2) 输出：对输入的一个排序$\{a_1', a_2', a_3', \cdots, a_n'\}$，且 $a_1' \leqslant a_2' \leqslant a_3' \leqslant \cdots \leqslant a_n'$。

(3) 有穷性：算法在执行 $O(n^2)$次后结束，这也是对算法性能的评估，即算法复杂度。

(4) 确定性：算法的每个步骤都是确定的。

(5) 可行性：算法的步骤能编程实现。

需要指出的是，上述第(5)条的可行性也是很重要的。有些算法并不能编程实现，例如一个有趣的排序算法——珠排序(Bead sort[①])，如果它用重力法，能够在 $O(1)$或 $O(\sqrt{n})$时间内得到排序结果，效率高到令人惊叹，但是无法编程实现。

视频讲解

2.3 算法的评估

上面已经反复提到，衡量算法性能的主要标准是时间复杂度，本节再针对算法竞赛展开说明。

为什么一般不讨论空间复杂度呢？在一般情况下，一个程序的空间复杂度是容易分析的，而时间复杂度往往关系到算法的根本逻辑，更能说明一个程序的优劣。因此，如果不特别说明，在提到"复杂度"时一般指时间复杂度。

注意，时间复杂度只是一个估计，并不需要精确计算。例如，在一个有 n 个数的无序数列中查找某个数 x，可能第一个数就是 x，也可能最后一个数才是 x，平均查找时间是 $n/2$ 次，但是把查找的时间复杂度记为 $O(n)$，而不是 $O(n/2)$。再如，冒泡排序算法的计算次数约等于 $n^2/2$ 次，但是仍记为 $O(n^2)$，而不是 $O(n^2/2)$。在算法分析中，规模 n 前面的常数系数被认为是不重要的。

还有，OJ 系统所判定的运行时间是整个程序运行所花的时间，而不是理论上算法所需要的时间。同一个算法，不同的人写出的程序，复杂度和运行时间可能差别很大，跟编程语言、逻辑结构、库函数等都有关系。

一个程序或算法的复杂度有以下可能。

1. $O(1)$

计算时间是一个常数，和问题的规模 n 无关。例如，用公式计算时，一次计算的复杂度就是 $O(1)$；哈希算法，用 hash 函数在常数时间内计算出存储位置；在矩阵 $A[M][N]$中查找第 i 行第 j 列的元素，只需要访问 $A[i][j]$就够了。

2. $O(\log_2 n)$

计算时间是对数，通常是以 2 为底的对数，每一步计算后，问题的规模减小一倍。例如

① https://en.wikipedia.org/wiki/Bead_sort.

在一个长度为 n 的有序数列中查找某个数，用折半查找的方法只需要 $\log_2 n$ 次就能找到。再如分治法，一般情况下，在每个步骤把规模减小一倍，所以一共有 $O(\log_2 n)$ 个步骤。

$O(\log_2 n)$ 和 $O(1)$ 没有太大差别。

3. $O(n)$

计算时间随规模 n 线性增长。在很多情况下，这是算法可能达到的最优复杂度，因为对输入的 n 个数，程序一般需要处理所有的数，即计算 n 次。例如查找一个无序数列中的某个数，可能需要检查所有的数。再如图问题，有 V 个点和 E 个边，大多数图的问题都需要搜索到所有的点和边，复杂度的上限就是 $O(V+E)$。

4. $O(n\log_2 n)$

这常常是算法能达到的最优复杂度。例如分治法，一共 $O(\log_2 n)$ 个步骤，每个步骤对每个数操作一次，所以总复杂度是 $O(n\log_2 n)$。用分治法思想实现的快速排序算法和归并排序算法的复杂度就是 $O(n\log_2 n)$。

5. $O(n^2)$

一个两重循环的算法，复杂度是 $O(n^2)$。例如冒泡排序是典型的两重循环。类似的复杂度有 $O(n^3)$、$O(n^4)$ 等。

6. $O(2^n)$

一般对应集合问题，例如一个集合中有 n 个数，要求输出它的所有子集，子集有 2^n 个。

7. $O(n!)$

在集合问题中，如果要求按顺序输出所有的子集，那么复杂度就是 $O(n!)$。

把上面的复杂度分成两类：①多项式复杂度，包括 $O(1)$、$O(n)$、$O(n\log_2 n)$、$O(n^k)$ 等，其中 k 是一个常数；②指数复杂度，包括 $O(2^n)$、$O(n!)$ 等。

如果一个算法是多项式复杂度，称它为“高效”算法；如果一个算法是指数复杂度，则称它为“低效”算法。可以这样通俗地解释“高效”和“低效”算法的区别：多项式复杂度的算法随着规模 n 的增加可以通过堆叠硬件来实现，“砸钱”是行得通的；而指数复杂度的算法增加硬件也无济于事，其增长的速度超出了人们的想象力。

竞赛题目一般的限制时间是 1s，对应普通计算机的计算速度是每秒千万次级，那么上述的时间复杂度可以换算出能解决问题的数据规模。例如，如果一个算法的复杂度是 $O(n!)$，当 $n=11$ 时，$11!=39\,916\,800$，这个算法只能解决 $n\leqslant 11$ 以内的问题。

下面的表 2.1 需要牢记。

表 2.1　问题规模和可用算法

问题规模 n	可用算法的时间复杂度					
	$O(\log_2 n)$	$O(n)$	$O(n\log_2 n)$	$O(n^2)$	$O(2^n)$	$O(n!)$
$n\leqslant 11$	√	√	√	√	√	√
$n\leqslant 25$	√	√	√	√	√	×
$n\leqslant 5000$	√	√	√	√	×	×
$n\leqslant 10^6$	√	√	√	×	×	×
$n\leqslant 10^7$	√	√	×	×	×	×
$n>10^8$	√	×	×	×	×	×

第 3 章　STL 和基本数据结构

✍容器
✍队列
✍栈
✍链表
✍set
✍map

STL(Standard Template Library)是 C++的标准模板库,竞赛中很多常用的数据结构、算法在 STL 中都有,熟练地掌握它们在很多题目中能极大地简化编程。本章所介绍的内容是竞赛训练的基本内容,需要完全掌握。

STL 包含容器(container)、迭代器(iterator)、空间配置器(allocator)、配接器(adapter)、算法(algorithm)、仿函数(functor) 6 个部分。本章介绍容器和两个常用算法。

3.1　容　　器

STL 容器包括顺序式容器和关联式容器。

1. 顺序式容器

顺序式容器包括 vector、list、deque、queue、priority_queue、stack 等,它们的特点如下。

- vector:动态数组,从末尾能快速插入与删除,直接访问任何元素。
- list:双链表,从任何地方快速插入与删除。
- deque:双向队列,从前面或后面快速插入与删除,直接访问任何元素。
- queue:队列,先进先出。
- priority_queue:优先队列,最高优先级元素总是第一个出列。
- stack:栈,后进先出。

2. 关联式容器

关联式容器包括 set、multiset、map、multimap 等。

- set:集合,快速查找,不允许重复值。
- multiset:快速查找,允许重复值。
- map:一对多映射,基于关键字快速查找,不允许重复值。
- multimap:一对多映射,基于关键字快速查找,允许重复值。

3.1.1　vector

数组是基本的数据结构，有静态数组和动态数组两种类型。在算法竞赛中，编码的惯例是：如果空间足够，能用静态数组就用静态数组，而不用指针管理动态数组，这样编程比较简单并且不会出错；如果空间紧张，可以用 STL 的 vector 建立动态数组，不仅节约空间，而且也不易出错。

视频讲解

vector[1] 是 STL 的动态数组，在运行时能根据需要改变数组大小。由于它以数组形式存储，也就是说它的内存空间是连续的，所以索引可以在常数时间内完成，但是在中间进行插入和删除操作会造成内存块的复制。另外，如果数组后面的内存空间不够，需要重新申请一块足够大的内存。这些都会影响 vector 的效率。

vector 容器是一个模板类，能存放任何类型的对象。

1. 定义

其示例如表 3.1 所示。

表 3.1　vector 定义示例

功　能	例　子	说　明
定义 int 型数组	vector < int > a;	默认初始化，a 为空
	vector < int > b(a);	用 a 定义 b
	vector < int > a(100);	a 有 100 个值为 0 的元素
	vector < int > a(100, 6);	100 个值为 6 的元素
定义 string 型数组	vector < string > a(10,"null");	10 个值为 null 的元素
	vector < string > vec(10,"hello");	10 个值为 hello 的元素
	vector < string > b(a.begin(), a.end());	b 是 a 的复制
定义结构型数组	struct point { int x, y; }; vector < point > a;	a 用来存坐标

用户还可以定义多维数组，例如定义一个二维数组：

```
vector < int > a[MAXN];
```

它的第一维大小是固定的 MAXN，第二维是动态的。用这个方式可以实现图的邻接表存储，细节见本书 10.2 节。

2. 常用操作

vector 的常用操作如表 3.2 所示。

表 3.2　vector 的常用操作

功　能	例　子	说　明
赋值	a.push_back(100);	在尾部添加元素
元素个数	int size=a.size();	元素个数

[1] http://www.cplusplus.com/reference/vector/vector/。

续表

功　能	例　子	说　明
是否为空	bool isEmpty=a.empty();	判断是否为空
打印	cout<<a[0]<<endl;	打印第一个元素
中间插入	a.insert(a.begin()+i, k);	在第 i 个元素前面插入 k
尾部插入	a.push_back(8);	尾部插入值为 8 的元素
尾部插入	a.insert(a.end(), 10,5);	尾部插入 10 个值为 5 的元素
删除尾部	a.pop_back();	删除末尾元素
删除区间	a.erase(a.begin()+i, a.begin()+j);	删除区间 $[i, j-1]$ 的元素
删除元素	a.erase(a.begin()+2);	删除第 3 个元素
调整大小	a.resize(n)	数组大小变为 n
清空	a.clear();	清空
翻转	reverse(a.begin(), a.end());	用函数 reverse()翻转数组
排序	sort(a.begin(), a.end());	用函数 sort()排序，从小到大排

下面用一个例题来说明 vector 的使用。

hdu 4841 "圆桌问题"

圆桌边围坐着 $2n$ 个人。其中 n 个人是好人，另外 n 个人是坏人。从第一个人开始数，数到第 m 个人，立即赶走该人；然后从被赶走的人之后开始数，再将数到的第 m 个人赶走，依此方法不断赶走围坐在圆桌边的人。

预先应如何安排这些好人与坏人的座位，才能使得在赶走 n 个人之后圆桌边围坐的剩余的 n 个人全是好人？

输入：多组数据，每组数据输入：$n, m \leq 32\,767$。

输出：对于每一组数据，输出 $2n$ 个大写字母，"G"表示好人，"B"表示坏人，50 个字母为一行，不允许出现空白字符。相邻数据间留有一个空行。

输入样例：

2 3

2 4

输出样例：

GBBG

BGGB

这个题目是约瑟夫问题。用 vector 模拟动态变化的圆桌，赶走 n 个人之后留下的都是好人。

程序如下：

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    vector<int> table;                          //模拟圆桌
    int n, m;
```

```
    while(cin >> n >> m){
        table.clear();
        for(int i = 0; i < 2 * n; i++)  table.push_back(i); //初始化
        int pos = 0;                                        //记录当前位置
        for(int i = 0; i < n; i++){                         //赶走 n 个人
            pos = (pos + m - 1) % table.size();             //圆桌是个环,取余处理
            table.erase(table.begin() + pos);               //赶走坏人,table 人数减 1
        }
        int j = 0;
        for(int i = 0; i < 2 * n; i++){                     //打印预先安排座位
            if(!(i % 50) && i)  cout << endl;               //50 个字母一行
            if(j < table.size() && i == table[j]){          //table 留下的都是好人
                j++;
                    cout <<"G";
            }
            else
                    cout <<"B";
        }
        cout << endl << endl;                               //留一个空行
    }
    return 0;
}
```

前面提到,vector 插入或者删除中间某一项时需要线性时间,即需要把这个元素后面的所有元素往后移或往前移,复杂度是 $O(n)$。如果频繁移动,则效率很低。hdu 4841 的 vector 程序用 erase()来删除中间元素就有这个问题。

3.1.2　栈和 stack[①]

栈是基本的数据结构之一,特点是“先进后出”。例如乘坐电梯时,先进电梯的最后出来;一盒泡腾片,最先放进盒子的药片位于最底层,最后被拿出来。

头文件:#include <stack>

栈的有关操作:

```
stack < Type > s;            //定义栈,Type 为数据类型,例如 int、float、char 等
s.push(item);                //把 item 放到栈顶
s.top();                     //返回栈顶的元素,但不会删除
s.pop();                     //删除栈顶的元素,但不会返回.在出栈时需要进行两步操作,即
                             //先 top()获得栈顶元素,再 pop()删除栈顶元素
s.size();                    //返回栈中元素的个数
s.empty();                   //检查栈是否为空,如果为空,返回 true,否则返回 false
```

下面用一个例子来说明栈的应用。

> **hdu 1062 “Text Reverse”**
>
> 翻转字符串。例如输入“olleh !dlrow”,输出“hello world!”。

① http://www.cplusplus.com/reference/stack/stack/。

用栈模拟，下面是程序：

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    int n;
    char ch;
    scanf("%d",&n);  getchar();
    while(n--){
        stack<char> s;
        while(true){
            ch = getchar();                 //一次读入一个字符
            if(ch == ' '||ch == '\n'||ch == EOF){
                while(!s.empty()){
                    printf("%c",s.top());  //输出栈顶
                    s.pop();               //清除栈顶
                }
                if(ch == '\n'||ch == EOF)  break;
                printf(" ");
            }
            else  s.push(ch);               //入栈
        }
        printf("\n");
    }
return 0;
}
```

爆栈问题。栈需要用空间存储，如果深度太大，或者存进栈的数组太大，那么总数会超过系统为栈分配的空间，这样就会爆栈，即栈溢出。其解决办法有下面两种：

（1）在程序中调大系统的栈，这种方法依赖于系统和编译器，竞赛的时候，在热身赛上可以试一试。

（2）手工写栈。有关内容见本书 10.5 节。

【习题】

比较复杂的用到栈的例子，请练习 hdu 1237“简单计算器”，逆波兰表达式。

3.1.3 队列和 queue[①]

队列是基本的数据结构之一，特点是“先进先出”。例如排队，先进队列的先得到服务。

头文件：#include <queue>

队列的有关操作：

```
queue<Type> q;              //定义栈,Type 为数据类型,例如 int、float、char 等
q. push(item);              //把 item 放进队列
q.front();                  //返回队首元素,但不会删除
q.pop();                    //删除队首元素
```

① http://www.cplusplus.com/reference/queue/queue/。

```
q.back();                               //返回队尾元素
q.size();                               //返回元素个数
q.empty();                              //检查队列是否为空
```

hdu 1702 "ACboy needs your help again!"

模拟栈和队列,栈是 FILO,队列是 FIFO。

分别用栈和队列模拟,下面是代码:

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int t,n,temp;
    cin>>t;
    while(t--){
        string str,str1;
        queue<int>Q;
        stack<int>S;
        cin>>n>>str;
        for(int i=0; i<n; i++){
            if(str=="FIFO"){          //队列
                cin>>str1;
                if(str1=="IN"){
                    cin>>temp;  Q.push(temp);
                }
                if(str1=="OUT"){
                    if(Q.empty()) cout<<"None"<<endl;
                    else{
                        cout<<Q.front()<<endl;
                        Q.pop();
                    }
                }
            }
            else{                     //栈
                cin>>str1;
                if(str1=="IN"){
                    cin>>temp;  S.push(temp);
                }
                if(str1=="OUT"){
                    if(S.empty()) cout<<"None"<<endl;
                    else {
                        cout<<S.top()<<endl;
                        S.pop();
                    }
                }
            }
        }
    }
    return 0;
}
```

3.1.4 优先队列和 priority_queue[①]

优先队列，顾名思义就是优先级最高的先出队。它是队列和排序的完美结合，不仅可以存储数据，还可以将这些数据按照设定的规则进行排序。每次的 push 和 pop 操作，优先队列都会动态调整，把优先级最高的元素放在前面。

优先队列的有关操作如下：

```
q.top();                    //返回具有最高优先级的元素值,但不删除该元素
q.pop();                    //删除最高优先级元素
q.push(item);               //插入新元素
```

在 STL 中，优先队列是用二叉堆来实现的，在队列中 push 一个数或 pop 一个数，复杂度都是 $O(\log_2 n)$。

可以用优先队列对数据排序：设定数据小的优先级高，把所有数 push 进优先队列后一个个 top 出来，就得到了从小到大的排序。其总复杂度是 $O(n\log_2 n)$。

图论的 Dijkstra 算法的程序实现用 STL 的优先队列能极大地简化代码，参考本书的 10.9.4 节。

【习题】

hdu 1873“看病要排队”。

3.1.5 链表和 list[②]

STL 的 list 是数据结构的双向链表，它的内存空间可以是不连续的，通过指针来进行数据的访问，它可以高效率地在任意地方删除和插入，插入和删除操作是常数时间的。

list 和 vector 的优缺点正好相反，它们的应用场景不同。

(1) vector：插入和删除操作少，随机访问元素频繁。

(2) list：插入和删除频繁，随机访问较少。

下面用一个例题来说明 list 的应用。

hdu 1276“士兵队列训练问题”

一队士兵报数：从头开始进行 1 至 2 报数，凡报到 2 的出列，剩下的向小序号方向靠拢，再从头开始进行 1 至 3 报数，凡报到 3 的出列，剩下的向小序号方向靠拢，以后从头开始轮流进行 1 至 2 报数、1 至 3 报数，直到剩下的人数不超过 3 为止。

输入：士兵人数。

输出：剩下士兵最初的编号。

① http://www.cplusplus.com/reference/queue/priority_queue/。

② http://www.cplusplus.com/reference/list/list/。

程序如下：

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    int t,n;
    cin >> t;
    while(t-- ) {
        cin >> n;
        int k = 2;
        list< int > mylist;                //定义
        list< int >::iterator it;
        for(int i = 1;i <= n;i++)
            mylist.push_back(i);           //赋值
        while(mylist.size() > 3) {
            int num = 1;
            for(it = mylist.begin(); it != mylist.end(); ){
                if(num++ % k == 0)
                    it = mylist.erase(it);
                else
                    it++;
            }
            k == 2 ? k = 3:k = 2;          //1 至 2 报数,1 至 3 报数
        }
        for(it = mylist.begin(); it != mylist.end(); it++){
            if (it != mylist.begin())
                cout << " ";
            cout << * it;
        }
        cout << endl;
    }
    return 0;
}
```

3.1.6 set[①]

set 就是集合。STL 的 set 用二叉搜索树实现，集合中的每个元素只出现一次，并且是排好序的。访问元素的时间复杂度是 $O(\log_2 n)$，非常高效。

set 和 3.1.7 节的 map 在竞赛题中的应用很广泛，特别是需要用二叉搜索树处理数据的题目，如果用 set 或 map 实现，能极大地简化代码。

set 的有关操作：

```
set< Type > A;                         //定义
A.insert(item);                        //把 item 放进 set
A.erase(item);                         //删除元素 item
```

① http://www.cplusplus.com/reference/set/set/。

```
A.clear();                          //清空 set
A.empty ();                         //判断是否为空
A.size();                           //返回元素个数
A.find(k);                          //返回一个迭代器,指向键值 k
A.lower_bound(k);                   //返回一个迭代器,指向键值不小于 k 的第一个元素
A.upper_bound();                    //返回一个迭代器,指向键值大于 k 的第一个元素
```

下面用一个例子来说明 set 的应用。

> **hdu 2094 "产生冠军"**
>
> 有一群人打乒乓球比赛,两两捉对厮杀,每两个人之间最多打一场比赛。
>
> 球赛的规则如下:
>
> 如果 A 打败了 B,B 又打败了 C,而 A 与 C 之间没有进行过比赛,那么就认定 A 一定能打败 C。
>
> 如果 A 打败了 B,B 又打败了 C,而且 C 又打败了 A,那么 A、B、C 三者都不可能成为冠军。
>
> 根据这个规则,无须循环较量,或许就能确定冠军。本题的任务就是对于一群比赛选手,在经过了若干场厮杀之后,确定是否已经产生了冠军。

这一题的思路是定义集合 A 和 B,把所有人放进集合 A,把所有有失败记录的放进集合 B。如果 $A-B=1$,则可以判断存在冠军,否则不能,请读者自己思考原因。

下面的程序演示了 set 的应用。

hdu 2094 程序

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    set<string> A, B;                       //定义集合
    string s1, s2;
    int n;
    while(cin >> n && n){
        for(int i = 0; i<n; i++) {
            cin >> s1 >> s2;
            A.insert(s1);  A.insert(s2);    //把所有人放进集合 A
            B.insert(s2);                   //把失败者放进集合 B
        }
        if(A.size() - B.size() == 1)
            cout << "Yes" << endl;
        else
            cout << "No" << endl;
        A.clear(); B.clear();
    }
    return 0;
}
```

3.1.7　map[①]

这里有一个常见的问题：有 n 个学生，每人有姓名 name 和学号 id，现在给定一个学生的 name，要求查找他的 id。

简单的做法是定义 string name[n]和 int id[n]（可以放在一个结构体中）存储信息，然后在 name[]中查找这个学生，找到后输出他的 id。这样做的缺点是需要搜索所有的 name[]，复杂度是 $O(n)$，效率很低。

利用 STL 中的 map 容器可以快速地实现这个查找，复杂度是 $O(\log_2 n)$。

map 是关联容器，它实现从键（key）到值（value）的映射。map 效率高的原因是它用平衡二叉搜索树来存储和访问。

在上述例子中，map 的具体操作如下。

（1）定义：map<string, int> student，存储学生的 name 和 id。

（2）赋值：例如 student["Tom"]=15。这里把"Tom"当成普通数组的下标来使用。

（3）查找：在找学号时，可以直接用 student["Tom"]表示他的 id，不用再去搜索所有的姓名。

map 用起来很方便。对于它的插入、查找、访问等操作，请读者自己阅读有关资料，并且认真掌握。

下面用一个例题来简单介绍 map 的使用。

hdu 2648 "Shopping"

女孩 dandelion 经常去购物，她特别喜欢一家叫"memory"的商店。由于春节快到了，所有商店的价格每天都在上涨。她想知道这家商店每天的价格排名。

输入：

第 1 行是数字 $n(n\leqslant 10\,000)$，代表商店的数量。

后面 n 行，每行有一个字符串（长度小于 31，只包含小写字母和大写字母），表示商店的名称。

然后一行是数字 $m(1\leqslant m\leqslant 50)$，表示天数。

后面有 m 部分，每部分有 n 行，每行是数字 s 和一个字符串 p，表示商店 p 在这一天涨价 s。

输出：包含 m 行，第 i 行显示第 i 天后店铺"memory"的排名。排名的定义为如果有 t 个商店的价格高于"memory"，那么它的排名是 $t+1$。

本题代码如下：

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int n, m, p;
```

① http://www.cplusplus.com/reference/map/map/。

```
    map<string, int> shop;
    while(cin >> n) {
        string s;
        for(int i=1; i<=n; i++) cin >> s;   //输入商店名字,实际上用不着处理
        cin >> m;
        while(m--) {
            for(int i=1; i<=n; i++) {
                cin >> p >> s;
                shop[s] += p;               //用 map 可以直接操作商店,加上价格
            }
            int rank = 1;
            map<string,int>::iterator it;   //迭代器
            for(it=shop.begin(); it != shop.end(); it++)
                if(it->second > shop["memory"])   //比较价格
                    rank++;
            cout << rank << endl;
        }
        shop.clear();
    }
    return 0;
}
```

3.2 sort()

STL 的排序函数 sort()①是算法竞赛中最常用的函数之一,它的定义有以下两种:

(1) void sort(RandomAccessIterator first, RandomAccessIterator last);

(2) void sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

返回值:无。

复杂度:$O(n\log_2 n)$。

注意,它排序的范围是[first, last),包括 first,不包括 last。

1. sort()的比较函数

排序是对比元素的大小。sort()可以用自定义的比较函数进行排序,也可以用系统的 4 种函数排序,即 less()、greater()、less_equal()、greater_equal()。在默认情况下,程序是按从小到大的顺序排序的,less()可以不写。

下面是程序例子。

```
#include<bits/stdc++.h>
using namespace std;
bool my_less(int i, int j)     {return (i < j);}  //自定义小于
bool my_greater(int i, int j)  {return (i > j);}  //自定义大于
int main(){
```

① http://www.cplusplus.com/reference/algorithm/sort/。

```
    vector<int> a = {3,7,2,5,6,8,5,4};
    sort(a.begin(),a.begin()+4);                //对前 4 个排序,输出 2 3 5 7 6 8 5 4
  //sort(a.begin(),a.end());                    //从小到大排序, 输出 2 3 4 5 5 6 7 8
  //sort(a.begin(),a.end(),less<int>());        //输出 2 3 4 5 5 6 7 8
  //sort(a.begin(),a.end(),my_less);            //自定义排序,输出 2 3 4 5 5 6 7 8
  //sort(a.begin(),a.end(),greater<int>());     //从大到小排序,输出 8 7 6 5 5 4 3 2
  //sort(a.begin(),a.end(),my_greater);         //输出 8 7 6 5 5 4 3 2
    for(int i=0; i<a.size(); i++)               //输出
        cout << a[i]<< " ";
    return 0;
}
```

sort()还可以对结构变量进行排序,例如:

```
struct Student{
    char name[256];
    int score;
};
bool compare(struct Student* a,struct Student* b){   //按分数从大到小排序
    return a->score > b->score;
}
…
vector<struct Student*> list;                  //定义 list,把学生信息存到 list 里
…
sort(list.begin(), list.end(), compare);       //按分数排序
```

2. 相关函数

stable_sort():当排序元素相等时,保留原来的顺序。在对结构体排序时,当结构体中的排序元素相等时,如果需要保留原序,可以用 stable_sort()。

partial_sort():局部排序。例如有 10 个数字,求最小的 5 个数。如果用 sort(),需要先全部排序,再输出前 5 个;而用 partial_sort()可以直接输出前 5 个。

3.3 next_permutation()

STL 提供求下一个排列组合的函数 next_permutation()①。例如 3 个字符 a、b、c 组成的序列,next_permutation()能按字典序返回 6 个组合,即 abc、acb、bac、bca、cab、cba。

函数 next_permutation()的定义有下面两种形式:

(1) bool next_permutation(BidirectionalIterator first, BidirectionalIterator last);

(2) bool next_permutation(BidirectionalIterator first, BidirectionalIterator last, Compare comp);

返回值:如果没有下一个排列组合,返回 false,否则返回 true。每执行 next_permutation()

① http://www.cplusplus.com/reference/algorithm/next_permutation/。

一次，就会把新的排列放到原来的空间里。

复杂度：$O(n)$。

注意，它排列的范围是[first, last)，包括 first，不包括 last。

在使用 next_permutation()的时候，初始序列一般是一个字典序最小的序列，如果不是，可以用 sort()排序，得到最小序列，然后再使用 next_permutation()，例题见本书的 4.1 节。

下面的例题是该函数的一个简单应用。

hdu 1027 "Ignatius and the Princess II"

给定 n 个数字，从 1 到 n，要求输出第 m 小的序列。

输入：数字 n 和 m，$1 \leqslant n \leqslant 1000$，$1 \leqslant m \leqslant 10\,000$。

输出：输出第 m 小的序列。

程序的思路是首先生成一个 $123 \cdots n$ 的最小字典序列，即初始序列，然后用 next_permutation()一个一个地生成下一个字典序更大的序列。

程序如下：

```
#include <bits/stdc++.h>
using namespace std;
int a[1001];
int main(){
    int n, m;
    while(cin >> n >> m){
        for(int i = 1; i <= n; i++) a[i] = i;     //生成一个字典序最小的序列
        int b = 1;
        do{
            if(b == m) break;
            b++;
        }while(next_permutation(a + 1,a + n + 1));
                                                   //注意第一个是 a + 1,最后一个是 a + n
        for(int i = 1; i < n; i++)                //输出第 m 大的字典序
            cout << a[i] << " ";
        cout << a[n] << endl;
    }
    return 0;
}
```

与 next_permutation()相关的函数如下：

- prev_permutation()：求前一个排列组合。
- lexicographical_compare()：字典比较。

【习题】

hdu 1716 "排列 2"。

第4章　搜索技术

- ✍递归和排列
- ✍子集生成和组合问题
- ✍BFS 和队列
- ✍A * 算法
- ✍DFS 和递归
- ✍八数码问题
- ✍回溯与剪枝
- ✍迭代加深搜索
- ✍IDA *

搜索是基本的编程技术，在算法竞赛学习中是基础的基础。搜索使用的算法是 BFS 和 DFS，BFS 用队列、DFS 用递归来具体实现。在 BFS 和 DFS 的基础上可以扩展出 A * 算法、双向广搜算法、迭代加深搜索、IDA * 等技术。本章详细介绍了这些知识点。

搜索技术是“暴力法”算法思想的具体实现。

人们常说：“要利用计算机强大的计算能力。”如果答案在一大堆数字里面，让计算机一个个去试，符合条件的不就是答案了吗？

没错，最基本的算法思想“暴力法”就是这样做的。例如，银行卡密码是 6 位数字，共 100 万个，对于计算机来说，尝试 100 万次只需要一瞬间。不过计算机也不是无敌的。为了应对计算机强大的计算能力，可以对密码进行强化设计。例如网络账号密码，大部分网站都要求长度在 8 位以上，并且混合数字、字母、标点等。从 40 多个符号中选 8 个组成密码，数量有 $40\times39\times38\times37\times36\times35\times34\times33>3$ 万亿，即使用计算机也不能很快算出来。

暴力法(Brute force，又译为蛮力法)：把所有可能的情况都罗列出来，然后逐一检查，从中找到答案。这种方法简单、直接，不玩花样，利用了计算机强大的计算能力。

虽然暴力法常常是低效的代名词，但是它仍然很有用，原因如下：

(1) 很多问题只能用暴力法解决，例如猜密码。

(2) 对于小规模的问题，暴力法完全够用，而且避免了高级算法需要的复杂编码，在竞赛中可以加快解题速度。在竞赛中也可以用暴力法来构造测试数据，以验证高级算法的正确性。

(3) 把暴力法当作参照(benchmark)。既然暴力法是“最差”的，那么可以把它当成一个比较来衡量另外的算法有多“好”。拿到题目后，如果没有其他思路，可以先试试暴力法，看是否能帮助产生灵感。

不过，在具体编程时常常需要对暴力法进行优化，以减少搜索空间，提高效率。例如利用剪枝技术跳过不符合要求的情况，从而减少复杂度。

虽然暴力搜索的思路很简单，但是操作起来并不容易。一般有以下操作：

(1) 找到所有可能的数据，并且用数据结构表示和存储。

(2) 剪枝。尽量多地排除不符合条件的数据，以减少搜索的空间。

(3) 用某个算法快速检索这些数据。

其中的第一步就可能很不容易。例如迷宫问题，如何列举从起点到终点的所有可能的路径[①]？再如图论中的“最短路径问题”，在地图上任取两个点，它们之间所有可行的路径可能是天文数字，以至于根本不能一一列举出来。所以计算最短路径的 Dijkstra 算法是用贪心法，进行从局部扩散到全局的搜索，不用列举所有可能的路径。

暴力法的主要操作是搜索，搜索的主要技术是 BFS 和 DFS。掌握搜索技术是学习算法竞赛的基础。在搜索时，具体的问题会有相应的数据结构，例如队列、栈、图、树等，读者应该能熟练地在这些数据结构上进行搜索的操作。

本章主要讲解 BFS 和 DFS，以及基于它们的优化技术，并以一些经典的搜索问题为例讲解算法思想，例如排列组合、生成子集、八皇后、八数码、图遍历等。

4.1 递归和排列

排列和组合问题是在暴力枚举的时候经常遇到的，一般有 3 种常见情况。

问题 4.1：打印 n 个数的全排列，共 $n!$ 个。

问题 4.2：打印 n 个数中任意 m 个数的全排列，共 $\frac{n!}{(n-m)!}$ 个。

问题 4.3：打印 n 个数中任意 m 个数的组合，共 $C_n^m=\frac{n!}{m!(n-m)!}$ 个。

本节用递归程序来实现问题 4.1 和问题 4.2，问题 4.3 将在下一节中讲解。

在计算机编程教材中都会提到递归的概念和应用，一般会用数学中的递推方程来讲解递归的概念，例如 $f(n)=f(n-1)+f(n-2)$。在计算机系统中，递归是通过嵌套来实现的，涉及指针、地址、栈的使用。

从算法思想上看，递归是把大问题逐步缩小，直到变成最小的同类问题的过程。例如 $n \to n-1 \to n-2 \to \cdots \to 1$，最后的小问题的解是已知的，一般是给定的初始条件。在递归的过程中，由于大问题和小问题的解决方法完全一样，那么大家自然可以想到，大问题的程序和小问题的程序可以写成一样。一个递归函数直接调用自己，就实现了程序的复用。

递归和分治法的思路非常相似，分治是把一个大问题分解为多个类型相同的子问题。事实上，一些涉及分治法的问题可以用递归来编程，典型的有快速排序、归并排序等。

对于编程初学者来说，递归是一个难以理解的编程概念，很容易绕晕。为了帮助理解，可以一步步打印出递归函数的输出，看它从大到小解决问题的过程。

编程竞赛中的暴力法常常需要考虑所有可能的情况，用递归编程可以轻松、方便地实现对搜索空间所有状态的遍历。

① 用 DFS 可以实现，程序也非常短。在学完本章之后，读者就能轻松地写出程序。

【问题 4.1】 打印 n 个数的全排列。

在用递归解决这个问题之前先给出 STL 的实现方法。

1. 用 STL 输出全排列

如果需要全排列的场景比较简单，可以直接用 C++ STL 的库函数 next_permutation()，它按字典序输出下一个排列。在使用之前，先用 sort()给数据排序，得到最小排列，然后每调用 next_permutation()一次，就得到一个大一点的排列。

next_permutation()的优点是能按从小到大的顺序输出排列。

```
#include <iostream>
#include <algorithm>                    //包含 sort()和 next_permutation()函数
using namespace std;
int main(){
    int data[4] = {5, 2, 1, 4};
    sort(data, data + 4);               //排序,得到最小排列
    do{
        for(int i = 0; i < 4; ++i)      //输出一个排列
            cout << data[i] << " ";
        cout << endl;
    }while(next_permutation(data, data + 4)); //把下一个排列放在 data 中
    return 0;
}
```

2. 用递归求全排列

下面用递归求全排列，代码很短，但是理解起来并不容易。读者可以自己打印每一个全排列的输出，然后认真理解。

在用递归之前，为了对比，先给出一个简单、粗暴的方法：以 10 个数的全排列为例，用排列组合的思路写一个 10 级的 for 循环，在每个 for 中选一个和前面的 for 用过的都不同的数。当 $n=10$ 时，一共有 $10!=3\ 628\ 800$ 个排列。

视频讲解

```
#include <bits/stdc++.h>
using namespace std;
int data[] = {7,1,2,3,4,5,6,8,9,10,12};         //本例子中用到前 10 个数
int main(){
    int num = 10;
    int i, j, k, m, n, p, q, r, s, t;           //10 个 for 循环
    for(i = 0; i < num; i++)
        for(j = 0; j < num; j++)
            if(j != i)                          //让 j 不等于 i
            for(k = 0; k < num; k++)
              if(k != j && k != i)              //让 k 不等于 i、j
                for(m = 0; m < num; m++)
                  if(m!= j && m!= i && m!= k)   //让 m 不等于 i、j、k
                      ...
              //最后打印出一个全排列: cout << data[i]<< data[j]...
}
```

上述的程序看起来很“笨”，下面用递归来写，显得很“美”。

用递归求全排列的思路：设定数字是{1 2 3 4 5…n}

(1) 让第 1 个数不同，得到 n 个数列。其办法是把第 1 个和后面的每个数交换。

1 2 3 4 5…n

2 1 3 4 5…n

⋮

n 2 3 4 5…1

以上 n 个数列，只要第 1 个数不同，不管后面的 $n-1$ 个数是怎么排列的，这 n 个数列都不同。

这是递归的第一层。

(2) 继续：在上面的每个数列中去掉第 1 个数，对后面的 $n-1$ 个数进行类似的排列。例如从上面第 2 行的{2 1 3 4 5…n}进入第二层(去掉首位 2)：

1 3 4 5…n

3 1 4 5…n

⋮

n 3 4 5…1

以上 $n-1$ 个数列，只要第 1 个数不同，不管后面的 $n-2$ 个数是怎么排列的，这 $n-1$ 个数列都不同。

这是递归的第二层。

(3) 重复以上步骤，直到用完所有数字。

在上面所有过程完成后，数列的总个数是 $n\times(n-1)\times(n-2)\cdots\times1=n!$。

递归打印全排列

```
#include<bits/stdc++.h>
using namespace std;
#define Swap(a, b) {int temp = a; a = b; b = temp;}
              //交换，也可以直接用 C++ STL 中的 swap()函数，但是速度慢一些
int data[] = {1,2,3,4,5,6,8,9,10,32,15,18,33};  //本例子中只用到前面 10 个数
int num = 0;                                     //统计全排列的个数，验证是不是 3628800
int Perm(int begin, int end){
    int i;
    if(begin == end) {                           //递归结束，产生一个全排列
                                                 //如果有必要，在此打印或处理这个全排列
        num++;                                   //统计全排列的个数
    }
    else
        for(i = begin; i <= end; i++) {
            Swap(data[begin], data[i]);          //把当前第 1 个数与后面的所有数交换位置
            Perm(begin + 1, end);
            Swap(data[begin], data[i]);          //恢复，用于下一次交换
        }
}
int main(){
    Perm(0, 9);                                  //求 10 个数的全排列
```

```
        cout << num << endl;                    //打印出排列总数,num = 10! = 3628800
    }
```

用这个程序可以检验普通计算机的计算能力。在上面的程序中加入 clock()统计时间：

```
#include <ctime>
int main() {
    clock_t start, end;
    start = clock();
    Perm(0, 9);
    end = clock();
    cout << (double)(end - start) / CLOCKS_PER_SEC << endl;
}
```

在作者的笔记本电脑上运行上述程序：

(1) Perm(0,9),计算 10 个数的全排列：10!=3 628 800,用时 0.055s。

(2) Perm(0,10),计算 11 个数的全排列：11!=39 916 800,用时 0.598s。

(3) Perm(0,11),计算 12 个数的全排列：12!=479 001 600,用时 7.305s。

12!/11!/10!的比值与 7.305s/0.598s/0.055s 的比值非常接近。

结论：笔记本电脑的计算能力大约是每秒千万次数量级。

竞赛题在一般情况下限时 1s,所以对于需要全排列的题目,其元素个数应该少于 11 个。

需要注意的是,从算法复杂度上看,上述两个程序的复杂度一样,都是 $O(n!)$。对于求全排列这样的问题,不可能有复杂度小于 $O(n!)$的算法,因为输出的数量就是 $n!$。在算法理论中,对必须要输出的元素进行的计数叫作“平凡下界”,这是程序运行所需要的最少花费。

上面的程序只要进行小的修改就能解决问题 4.2。

【问题 4.2】 打印 n 个数中任意 m 个数的全排列。

例如在 10 个数中取任意 3 个数的全排列,在 Perm()中只修改一个地方就可以了：

```
if(begin == 3) {                          //把 Perm()函数中的 end 改为 3 即可,其他都不变
    cout << data[0]<< data[1]<< data[2]<< endl;//打印 10 个数中 3 个数的全排列
    num++;                                //统计全排列的个数,应该是 10×9×8 = 720 个
}
```

【问题 4.3】 打印 n 个数中任意 m 个数的组合。

问题 4.3 和问题 4.1 的区别为排列是有序的,组合是无序的。其中一个特例是在 n 个数中取 n 个数的组合,只有 1 种情况,就是这 n 个数本身。

问题 4.3 将在 4.2 节中讲解。

4.2　子集生成和组合问题

在 4.1 节求 10 个数的排列问题中,如果不需要输出全排列,而是输出组合,即子集(子集内部的元素是没有顺序的),那么该如何做呢?

一个包含 n 个元素的集合$\{a_0, a_1, a_2, a_3, \cdots, a_{n-1}\}$，它的子集有$\{\phi\}$，$\{a_0\}$，$\{a_1\}$，$\{a_2\}$，…，$\{a_0, a_1, a_2\}$，…，$\{a_0, a_1, a_2, a_3, \cdots, a_{n-1}\}$，共 2^n 个。

用二进制的概念进行对照是最直观的。

例如 $n=3$ 的集合$\{a_0, a_1, a_2\}$，它的子集和二进制数的对应关系如表 4.1 所示。

表 4.1　$n=3$ 的集合$\{a_0, a_1, a_2\}$的子集和二进制数的对应关系

子　　集	ϕ	a_0	a_1	a_1, a_0	a_2	a_2, a_0	a_2, a_1	a_2, a_1, a_0
二进制数	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1

所以，每个子集对应一个二进制数，这个二进制数中的每个 1 都对应着这个子集中的某个元素，而且子集中的元素是没有顺序的。

从这个表也可以理解为什么子集的数量是 2^n 个，因为所有二进制数的总个数是 2^n。

下面的程序通过处理每个二进制数中的 1 打印出了所有的子集。

```
#include<bits/stdc++.h>
using namespace std;
void print_subset(int n){
    for(int i=0;i<(1<<n);i++) {
     //i:0～2ⁿ,每个 i 的二进制数对应一个子集,一次打印一个子集,最后得到所有子集
        for(int j=0;j<n;j++)          //打印一个子集,即打印 i 的二进制数中所有的 1
            if(i & (1<<j))            //从 i 的最低位开始逐个检查每一位,如果是 1,打印
                cout<<j<<" ";
        cout<<endl;
    }
}
int main(){
    int n;
    cin>>n;                           //n:集合中元素的总数量
    print_subset(n);                  //打印所有的子集
}
```

回到问题 4.3：打印 n 个数中任意 m 个数的组合。对照子集生成的二进制方法，已经知道一个子集对应一个二进制数。那么一个有 k 个元素的子集，它对应的二进制数中有 k 个 1。所以，问题就转化为查找 1 的个数为 k 的二进制数，这些二进制数就是需要打印的子集。

那么如何判断二进制数中 1 的个数为 k①？简单的方法是对这个 n 位二进制数逐位检查，共需要检查 n 次。

另外有一个更快的方法，它可以直接定位二进制数中 1 的位置，跳过中间的 0。它用到一个神奇的操作——$kk=kk \& (kk-1)$，功能是消除 kk 的二进制数的最后一个 1。连续进行这个操作，每次消除一个 1，直到全部消除为止，操作次数就是 1 的个数。例如二进制数 1011，经过连续 3 次操作后，所有的 1 都消除了：

① glibc 有处理二进制数的内部函数，其中 int __builtin_popcount(unsigned int x)直接返回 x 中 1 的个数。

$$1011\ \&\ (1011-1)=1011\ \&\ 1010=1010$$

$$1010\ \&\ (1010-1)=1010\ \&\ 1001=1000$$

$$1000\ \&\ (1000-1)=1000\ \&\ 0111=0000$$

利用这个操作可以计算出二进制数中 1 的个数。用 num 统计 1 的个数，具体步骤如下：

(1) 用 $kk=kk\ \&\ (kk-1)$清除 kk 的最后一个 1。

(2) num++。

(3) 继续上述操作，直到 $kk=0$。

在树状数组中也有一个类似的操作——lowbit(x)=$x\ \&\ -x$，功能是计算 x 的二进制数的最后一个 1。

下面的程序在子集生成程序的基础上实现了问题 4.3 的要求：

```
#include<bits/stdc++.h>
using namespace std;
void print_set(int n,int k){
    for(int i = 0; i<(1<<n); i++){
        int num = 0, kk = i;          //num 统计 i 中 1 的个数;kk 用来处理 i
        while(kk){
            kk = kk&(kk-1);           //清除 kk 中的最后一个 1
            num++;                    //统计 1 的个数
        }
        if(num == k){                 //二进制数中的 1 有 k 个,符合条件
            for(int j = 0; j<n; j++)
                if(i & (1<<j))
                    cout << j << " ";
            cout << endl;
        }
    }
}
int main(){
    int n, k;                         //n:元素的总数量; k:个数为 k 的子集
    cin >> n >> k;
    print_set(n,k);
}
```

4.3　BFS

4.3.1　BFS 和队列

深度优先搜索(Depth-First Search，DFS)和广度优先搜索(Breadth-First Search，BFS，或称为宽度优先搜索)是基本的暴力技术，常用于解决图、树的遍历问题。

首先考虑算法思路。以老鼠走迷宫为例，这是 DFS 和 BFS 在现实中的模型。迷宫内部的路错综复杂，老鼠从入口进去后怎么才能找到出口？有两种不同的方法：

(1) 一只老鼠走迷宫。它在每个路口都选择先走右边(当然,选择先走左边也可以),能走多远就走多远,直到碰壁无法继续往前走,然后回退一步,这一次走左边,接着继续往下走。用这个办法能走遍**所有**的路,而且**不会重复**(这里规定回退不算重复走)。这个思路就是 DFS。

(2) 一群老鼠走迷宫。假设老鼠是无限多的,这群老鼠进去后,在每个路口派出部分老鼠探索所有没走过的路。走某条路的老鼠,如果碰壁无法前行,就停下;如果到达的路口已经有其他老鼠探索过了,也停下。很显然,**所有**的道路都会走到,而且**不会重复**。这个思路就是 BFS。BFS 看起来像"并行计算",不过,由于程序是单机顺序运行的,所以可以把 BFS 看成是并行计算的模拟。

在具体编程时,一般用队列这种数据结构来具体实现 BFS,甚至可以说"BFS=队列";对于 DFS,也可以说"DFS=递归",因为用递归实现 DFS 是最普遍的。DFS 也可以用"栈"这种数据结构来直接实现,栈和递归在算法思想上是一致的。

下面用一个图遍历的题目来介绍 BFS 和队列。

hdu 1312 "Red and Black"

有一个长方形的房间,铺着方形瓷砖,瓷砖为红色或黑色。一个人站在黑色瓷砖上,他可以按上、下、左、右方向移动到相邻的瓷砖。但他不能在红色瓷砖上移动,只能在黑色瓷砖上移动。编程计算他可以到达的黑色瓷砖的数量。

输入:第 1 行包含两个正整数 W 和 H,W 和 H 分别表示 x 方向和 y 方向上的瓷砖数量。W 和 H 均不超过 20。下面有 H 行,每行包含 W 个字符。每个字符表示一片瓷砖的颜色。用符号表示如下:"•"表示黑色瓷砖;"#"表示红色瓷砖;"@"代表黑色瓷砖上的人,在数据集中只出现一次。

输出:一个数字,这个人从初始瓷砖能到达的瓷砖总数量(包括起点)。

这个题目跟老鼠走迷宫差不多:"#"相当于不能走的陷阱或墙壁,"•"是可以走的路。下面按"一群老鼠走迷宫"的思路编程。

要遍历所有可能的点,可以这样走:从起点 1 出发,走到它所有的邻居 2、3;逐一处理每个邻居,例如在邻居 2 上,再走它的所有邻居 4、5、6;继续以上过程,直到所有点都被走到,如图 4.1 所示。这是一个"扩散"的过程,如果把搜索空间看成一个池塘,丢一颗石头到起点位置,激起的波浪会一层层扩散到整个空间。需要注意的是,扩散按从近到远的顺序进行,因此,从每个被扩散到的点到起点的路径都是最短的。这个特征对解决迷宫这样的最短路径问题很有用。

用队列来处理这个扩散过程非常清晰、易懂,对照图 4.1:

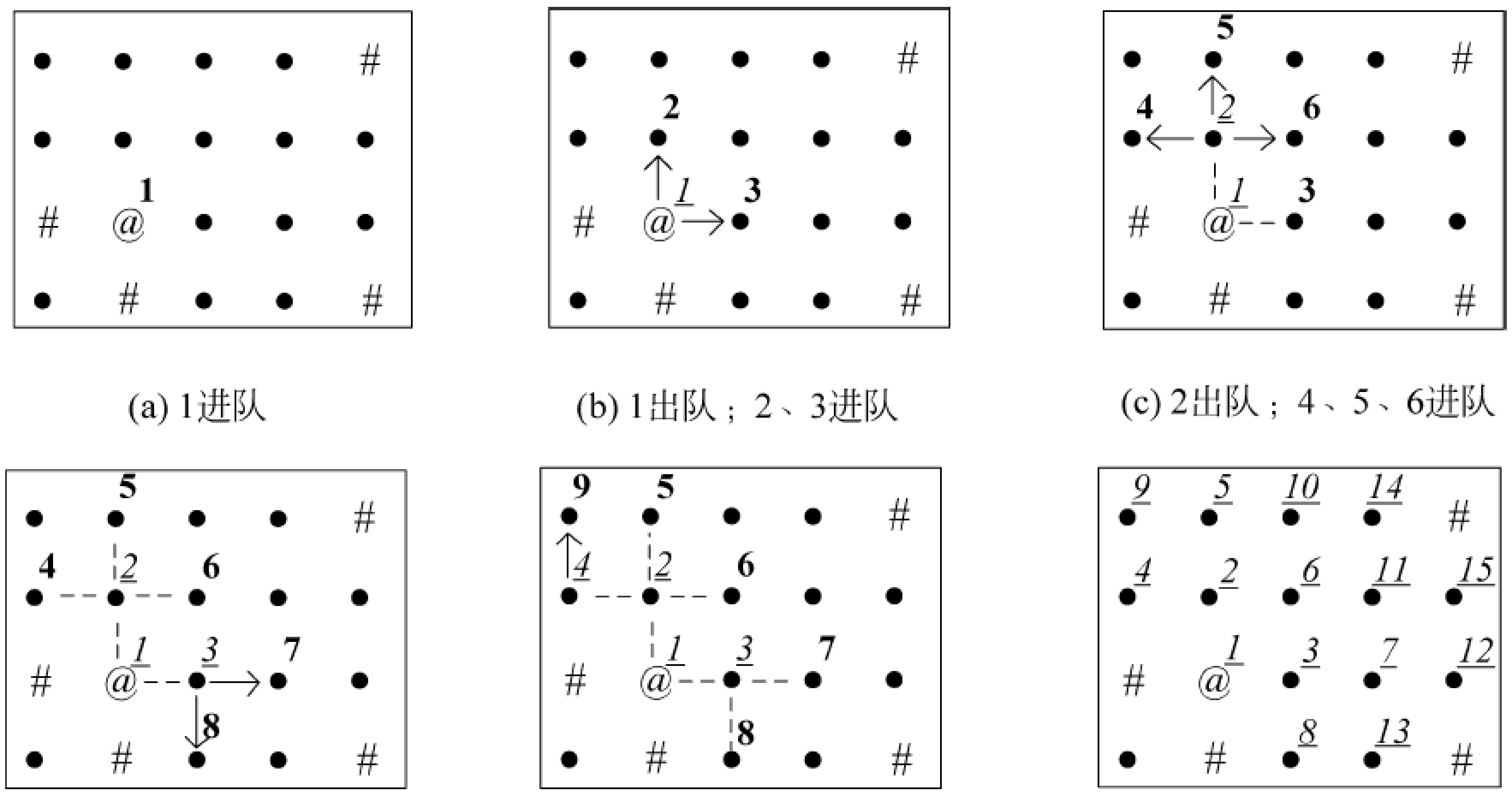
(a) 1 进队。当前队列是{1}。

(b) 1 出队,1 的邻居 2、3 进队。当前队列是{2,3}(可以理解为从 1 扩散到 2、3)。

(c) 2 出队,2 的邻居 4、5、6 进队。当前队列是{3,4,5,6}(可以理解为从 2 扩散到 4、5、6)。

(d) 3 出队,7、8 进队。当前队列是{4,5,6,7,8}(可以理解为从 3 扩散到 7、8)。

(e) 4 出队,9 进队。当前队列是{5,6,7,8,9}。



(a) 1进队　(b) 1出队；2、3进队　(c) 2出队；4、5、6进队

(d) 3出队；7、8进队　(e) 4出队；9进队　(k) 最后结果

图 4.1　BFS 过程

(f) 5 出队,10 进队。当前队列是{6,7,8,9,10}。

(g) 6 出队,11 进队。当前队列是{7,8,9,10,11}。

(h) 7 出队,12、13 进队。当前队列是{8,9,10,11,12,13}。

(i) 8、9 出队,10 出队,14 进队。当前队列是{11,12,13,14}。

(j) 11 出队,15 进队。当前队列是{12,13,14,15}。

(k) 12、13、14、15 出队。当前队列是空{},结束。

hdu 1312 题的 BFS 程序

```
#include <bits/stdc++.h>
using namespace std;
char room[23][23];
int dir[4][2] = {
    {-1,0},                          //向左.左上角的坐标是(0, 0)
    {0, -1},                         //向上
    {1,0},                           //向右
    {0,1}                            //向下
};
int Wx, Hy, num;                     //Wx 行,Hy 列.用 num 统计可走的位置有多少
#define CHECK(x, y) (x<Wx && x>=0 && y>=0 && y<Hy)     //是否在 room 中
struct node {int x,y;};
void BFS(int dx,int dy){
    num = 1;                         //起点也包含在砖块内
    queue<node> q;                   //队列中放坐标点
    node start, next;
    start.x = dx;
    start.y = dy;
    q.push(start);
    while(!q.empty()) {
        start = q.front();
```

```
            q.pop();
            //cout <<"out"<< start.x << start.y << endl;    //打印出队列情况,进行验证
            for(int i = 0; i < 4; i++) {                     //按左、上、右、下 4 个方向顺时针逐一搜索
                next.x = start.x + dir[i][0];
                next.y = start.y + dir[i][1];
                if(CHECK(next.x,next.y) && room[next.x][next.y] == '.') {
                    room[next.x][next.y] = '#';             //进队之后标记为已经处理过
                    num++;
                    q.push(next);
                }
            }
        }
}
int main(){
    int x, y, dx, dy;
    while (cin >> Wx >> Hy) {                               //Wx 行,Hy 列
        if (Wx == 0 && Hy == 0)                             //结束
            break;
        for (y = 0; y < Hy; y++) {                          //有 Hy 列
            for (x = 0; x < Wx; x++) {                      //一次读入一行
                cin >> room[x][y];
                if(room[x][y] == '@') {                     //读入起点
                    dx = x;
                    dy = y;
                }
            }
        }
        num = 0;
        BFS(dx, dy);
        cout << num << endl;
    }
    return 0;
}
```

【习题】

视频讲解

poj 3278 "Catch That Cow"。

poj 1426 "Find The Multiple"。

poj 3126 "Prime Path"。

poj 3414 "Pots"。

hdu 1240 "Asteroids!"。

hdu 4460 "Friend Chains"。

4.3.2　八数码问题和状态图搜索

BFS 搜索处理的对象不仅可以是一个数,还可以是一种"状态"。八数码问题是典型的状态图搜索问题。

1. 八数码问题

在一个 3×3 的棋盘上放置编号为 1～8 的 8 个方块，每个占一格，另外还有一个空格。与空格相邻的数字方块可以移动到空格里。任务 1：指定初始棋局和目标棋局(如图 4.2 所示)，计算出最少的移动步数；任务 2：输出数码的移动序列。

把空格看成 0，一共有 9 个数字。

输入样例：

1 2 3 0 8 4 7 6 5

1 0 3 8 2 4 7 6 5

输出样例：

2

1	2	3
	8	4
7	6	5

1		3
8	2	4
7	6	5

图 4.2　初始棋局和目标棋局

把一个棋局看成一个状态图，总共有 9！=362 880 个状态。从初始棋局开始，每次移动转到下一个状态，到达目标棋局后停止。

八数码问题是一个经典的 BFS 问题。前面章节中提到 BFS 是从近到远的扩散过程，适合解决最短距离问题。八数码从初始状态出发，每次转移都逐步逼近目标状态。每转移一次，步数加一，当到达目标时，经过的步数就是最短路径。

图 4.3 是样例的转移过程。该图中起点为(*A*，0)，*A* 表示状态，即{1 2 3 0 8 4 7 6 5}这个棋局；0 是距离起点的步数。从初始状态 *A* 出发，移动数字 0 到邻居位置，按左、上、右、下的顺时针顺序，有 3 个转移状态 *B*、*C*、*D*；目标状态是 *F*，停止。

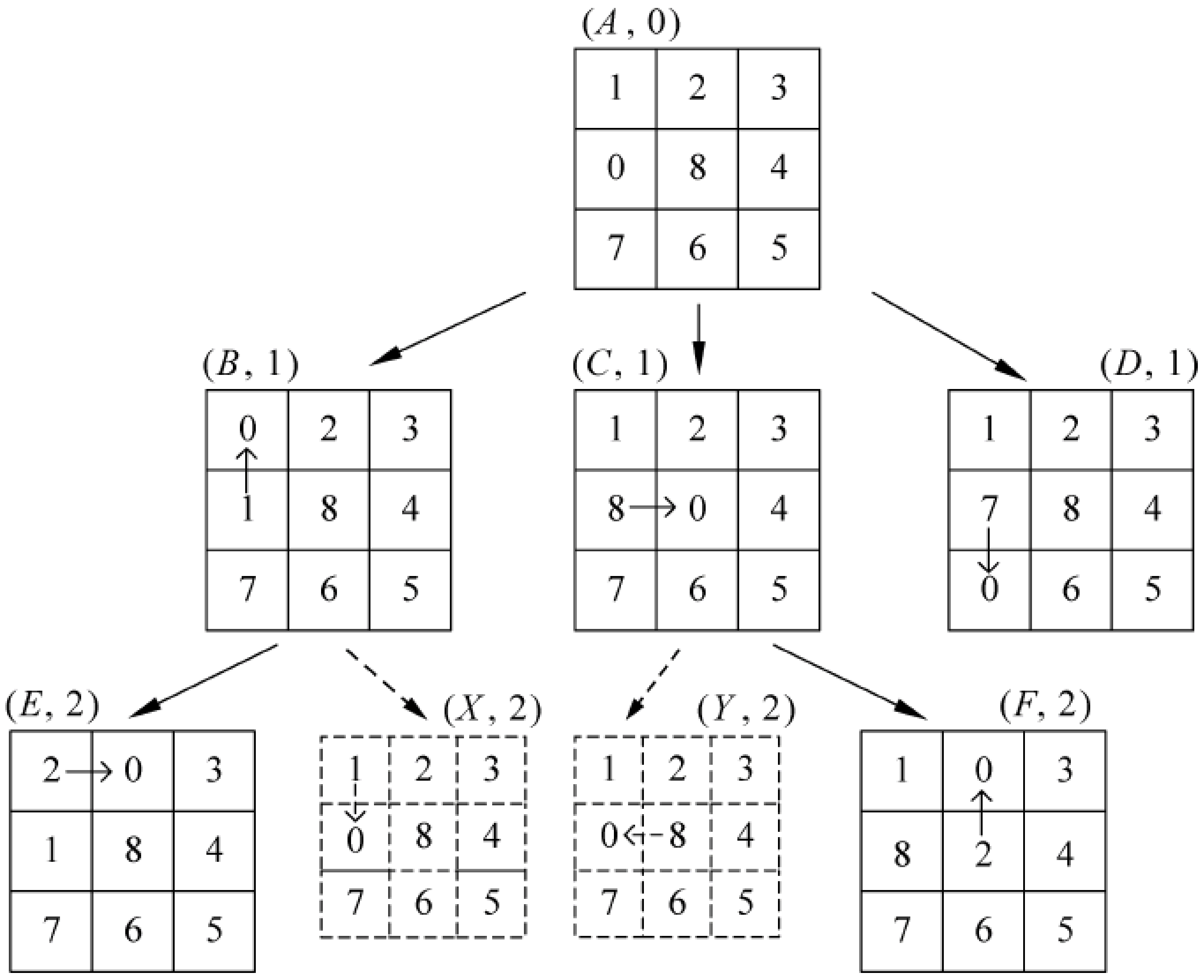


图 4.3　八数码问题的搜索树

用队列描述这个 BFS 过程：

(1) *A* 进队，当前队列是{*A*}；

(2) *A* 出队，*A* 的邻居 *B*、*C*、*D* 进队，当前队列是{*B*，*C*，*D*}，步数为 1；

(3) *B* 出队，*E* 进队，当前队列是{*C*，*D*，*E*}，*E* 的步数为 2；

(4) *C* 出队，转移到 *F*，检验 *F* 是目标状态，停止，输出 *F* 的步数 2。

仔细分析上述过程，发现从 B 状态出发实际上有 E、X 两个转移方向，而 X 正好是初始状态 A，重复了。同理 Y 状态也是重复的。如果不去掉这些重复的状态，程序会产生很多无效操作，复杂度大大增加。因此，八数码的重要问题其实是判重。

如果用暴力的方法判重，每次把新状态与 9！＝362 880 个状态对比，可能有 9！×9！次检查，不可行。因此需要一个快速的判重方法。

本题可以用数学方法“康托展开(Cantor Expansion)”来判重。

2. 康托展开

康托展开是一种特殊的哈希函数。在本题中，康托展开完成了如表 4.2 所示的工作：

表 4.2　本题中康托展开完成的工作

状　　态	012345678	012345687	012345768	012345786	…	876543210
Cantor	0	1	2	3	…	362 880－1

第 1 行是 0～8 这 9 个数字的全排列，共 9！＝362 880 个，按从小到大排序。第 2 行是每个排列对应的位置，例如最小的{012345678}在第 0 个位置，最大的{876543210}在最后的 362 880－1 这个位置。

函数 Cantor()实现的功能是：输入一个排列，即第 1 行的某个排列，计算出它的 Cantor 值，即第 2 行对应的数。

Cantor()的复杂度为 $O(n^2)$，n 是集合中元素的个数。在本题中，完成搜索和判重的总复杂度是 $O(n!n^2)$，远比用暴力判重的总复杂度 $O(n!n!)$小。

有了这个函数，八数码的程序能很快判重：每转移到一个新状态，就用 Cantor()判断这个状态是否处理过，如果处理过，则不转移。

下面举例讲解康托展开的**原理**。

例子：判断 2143 是{1，2，3，4}的全排列中第几大的数。

计算排在 2143 前面的排列数目，可以将问题转换为以下排列的和：

(1) 首位小于 2 的所有排列。比 2 小的只有 1 一个数，后面 3 个数的排列有 3×2×1＝3！个(即 1234、1243、1324、1342、1423、1432)，写成 1×3！＝6。

(2) 首位为 2、第 2 位小于 1 的所有排列。无，写成 0×2！＝0。

(3) 前两位为 21、第 3 位小于 4 的所有排列。只有 3 一个数(即 2134)，写成 1×1！＝1。

(4) 前 3 位为 214、第 4 位小于 3 的所有排列。无，写成 0×0！＝0。

求和：1×3！＋0×2！＋1×1！＋0×0！＝7，所以 2143 是第 8 大的数。如果用 int visited[24]数组记录各排列的位置，{2143}就是 visited[7]；第一次访问这个排列时，置 visited[7]＝1；当再次访问这个排列的时候发现 visited[7]等于 1，说明已经处理过，判重。

根据上面的例子得到康托展开公式。

把一个集合产生的全排列按字典序排序，第 X 个排列的计算公式如下：

$$X=a[n]\times(n-1)!+a[n-1]\times(n-2)!+\cdots+a[i]\times(i-1)!+\cdots+a[2]\times 1!+a[1]\times 0![1]$$

其中，$a[i]$为当前未出现的元素排在第几个(从 0 开始)，并且有 $0\leqslant a[i]<i\ (1\leqslant i\leqslant n)$。

上述过程的反过程是康托逆展开：某个集合的全排列，输入一个数字 k，返回第 k 大的

排列。

下面的程序用“BFS+Cantor”解决了八数码问题，其中 BFS 用 STL 的 queue 实现[①]。

```
#include <bits/stdc++.h>
const int LEN = 362880;              //状态共 9!= 362 880 种
using namespace std;
struct node{
    int state[9];                    //记录一个八数码的排列,即一个状态
    int dis;                         //记录到起点的距离
};

int dir[4][2] = {{-1,0}, {0, -1},{1,0},{0,1}};
                                     //左、上、右、下顺时针方向.左上角的坐标是(0,0)
int visited[LEN] = {0};              //与每个状态对应的记录,Cantor()函数对它置数,并判重
int start[9];                        //开始状态
int goal[9];                         //目标状态
long int factory[] = {1,1,2,6,24,120,720,5040,40320,362880};
                                     //Cantor()用到的常数
bool Cantor(int str[], int n) {      //用康托展开判重
    long result = 0;
    for(int i = 0; i < n; i++) {
        int counted = 0;
        for(int j = i+1; j < n; j++) {
            if(str[i] > str[j])      //当前未出现的元素排在第几个
                ++counted;
        }
        result += counted * factory[n-i-1];
    }
    if(!visited[result]) {           //没有被访问过
        visited[result] = 1;
        return 1;
    }
    else
        return 0;
}
int bfs() {
    node head;
    memcpy(head.state, start, sizeof(head.state));    //复制起点的状态
    head.dis = 0;
    queue<node> q;                   //队列中的内容是记录状态
    Cantor(head.state, 9);           //用康托展开判重,目的是对起点的 visited[]赋初值
    q.push(head);                    //第一个进队列的是起点状态

    while(!q.empty()) {              //处理队列
        head = q.front();
        q.pop();                     //可在此处打印 head.state,看弹出队列的情况
        int z;
        for(z = 0; z < 9; z++)       //找这个状态中元素 0 的位置
            if(head.state[z] == 0)//找到了
```

① 本题中的队列比较简单，如果不用 STL，也可以用简单的方法模拟队列，请搜索网上的代码。

```
                break;
            //转化为二维,左上角是原点(0,0)
        int x = z%3;                        //横坐标
        int y = z/3;                        //纵坐标
        for(int i = 0; i<4; i++){           //上、下、左、右最多可能有 4 个新状态
            int newx = x+dir[i][0];         //元素 0 转移后的新坐标
            int newy = y+dir[i][1];
            int nz = newx + 3*newy;         //转化为一维
            if(newx>=0 && newx<3 && newy>=0 && newy<3) {    //未越界
                node newnode;
                memcpy(&newnode,&head,sizeof(struct node));  //复制这新的状态
                swap(newnode.state[z], newnode.state[nz]);   //把 0 移动到新的位置
                newnode.dis ++;
                if(memcmp(newnode.state,goal,sizeof(goal)) == 0)
                                              //与目标状态对比
                    return newnode.dis;       //到达目标状态,返回距离,结束
                if(Cantor(newnode.state, 9))  //用康托展开判重
                    q.push(newnode);          //把新的状态放进队列
            }
        }
    }
    return -1;                                //没找到
}
int main(){
    for(int i = 0; i<9; i++)  cin>>start[i];  //初始状态
    for(int i = 0; i<9; i++)  cin>>goal[i];   //目标状态
    int num = bfs();
    if(num != -1)  cout<<num<<endl;
    else           cout<<"Impossible"<<endl;
    return 0;
}
```

上述代码的细节很多,请读者仔细体会,要求能独立写出来。

15 数码问题。八数码问题只有 9!种状态,对于更大的问题,例如 4×4 棋盘的 15 数码问题,有 $16! \approx 2\times10^{13}$ 种状态,如果仍然用数组存储状态,远远不够,此时需要更好的算法①。

【习题】

poj 1077 “Eight”,八数码问题。另外,在学过下一节的 A* 算法后可重新做这道题。

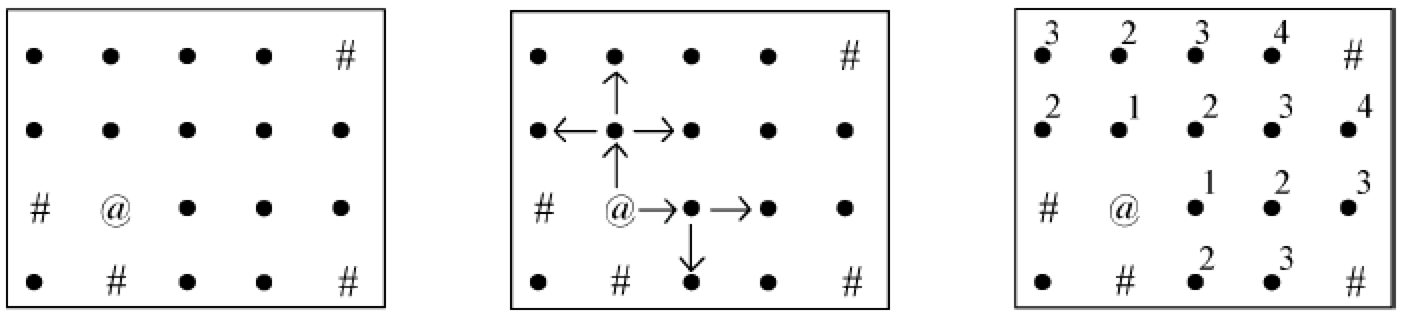
4.3.3 BFS 与 A* 算法

1. 用 BFS 求最短路径

最短路径是图论的一个基本问题,有很多复杂的算法。不过,在特殊的地图中,BFS 也是很好的最短路径算法。下面仍然以 hdu 1312"Red and Black"的方格图为例,任务是求两点之间的最短路径。

① 八数码的多种解法,例如双向广搜、A*、IDA* 等,请参考“https://www.cnblogs.com/zufezzt/p/5659276.html”(永久网址: perma.cc/YV2V-GT6C)。

在图 4.4 中,黑点"•"表示可以走的路,"#"表示不能走。求起点"@"到所有黑点"•"的最短距离。



(a) 迷宫图　　(b) BFS过程　　(c) 最短距离

图 4.4　用 BFS 找最短路径

方法很简单,从"@"出发,用 BFS 搜索所有点,记录到达每个点时经过的步数,即可得到从"@"到所有黑点的最短距离,图 4.4(c)标出了结果。

在这个例子中,BFS 搜最短路径的计算复杂度是 $O(V+E)$,非常好。

这个例子很特殊,图是方格形的,相邻两点之间的距离相同。也就是说,绕路肯定更远;BFS 先扩展到的路径,距离肯定是最短的。

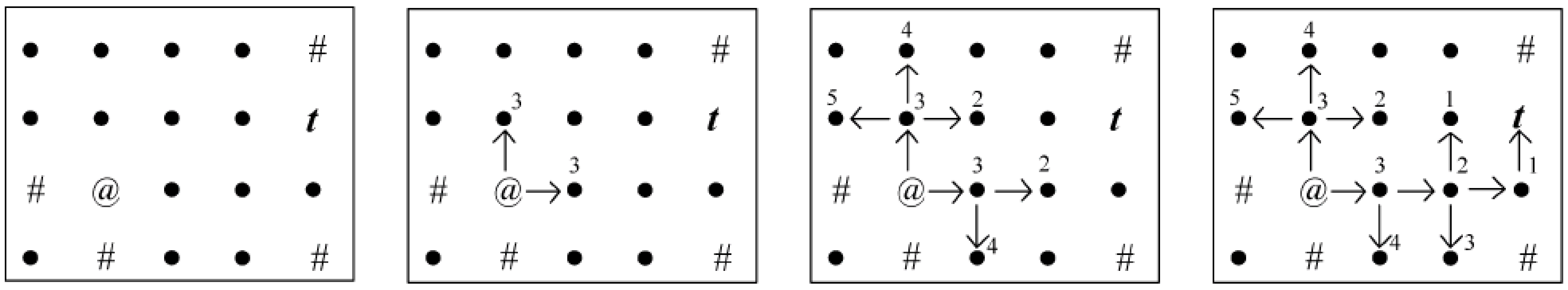
如果相邻点的距离不同,绕路可能更近,BFS 就不适用了。关于最短路径的通用算法,请阅读本书的 10.9 节的内容。

下面的 A * 算法是 BFS 的优化。

2. A * 算法与最短路径

BFS 是一种"盲目的"搜索技术,它在搜索的过程中并不理会目标在哪里,只顾自己乱走,当然最后总会到达终点。

稍微改变 hdu 1312 的方格图,见图 4.5(a),现在的任务是求起点"@"到终点"*t*"的最短路径。



(a) 起点和终点　　(b) 第1轮搜索　　(c) 第2轮搜索　　(d) 最短距离

图 4.5　启发式搜索

如果仍然用 BFS 求解,程序会搜索所有的点,直到遇到 *t* 点。不过,如果让一个人走这个图,他会一眼看出向右上方走可以更快地找到到达 *t* 的最短路径。人有"智能",那么能否把这种智能教给程序呢？这就是"启发式"搜索算法。启发式搜索算法有很多种,A * 算法是其中比较简单的一种。

简单地说,A * 算法是"BFS+贪心"①。有关贪心法的解释,请阅读本书的 6.1 节。

在图 4.5(a)中,程序如何知道向右上方走能更快地到达 *t*?这里引入曼哈顿距离的概念。曼哈顿距离是指两个点在标准坐标系上的实际距离,在图 4.5 中就是@的坐标和 *t* 的

① 这个网页用动画演示了 BFS、A * 、Dijkstra 算法的原理,并给出了比较详细的伪代码描述,非常值得一看,网址为 https://www.redblobgames.com/pathfinding/a-star/introduction.html(永久网址:https://perma.cc/N2DB-5LDY)。

坐标在横向和纵向的距离之和，它也被形象地称为“出租车距离”。

图 4.5(b)是从起点开始的第 1 轮 BFS 搜索，邻居点上标注的数字 3 是这个点到终点 t 的曼哈顿距离。图 4.5(c)是第 2 轮搜索，标注 2 的点是离终点更近的点，从这些点继续搜索；标注 4 和 5 的点距离终点远，先暂时停止搜索。经过多轮搜索，最后到达了终点 t，如图 4.5(d)所示。

在这个过程中，图中很多“不好的”点并不需要搜索到，从而优化了搜索过程。

上面的图例比较简单，如果起点和终点之间有很多障碍，搜索范围也会沿着障碍兜圈子，之后才能到达终点，不过，仍然有很多点不需要搜索。以下面的图 4.6 为例，A 是起点，B 是终点，黑色方块是障碍，浅色阴影方块是用曼哈顿距离进行启发式搜索所经过的部分，其他无色方块是不需要搜索的。搜索结束后，得到一条最短路径，见图中的虚线。

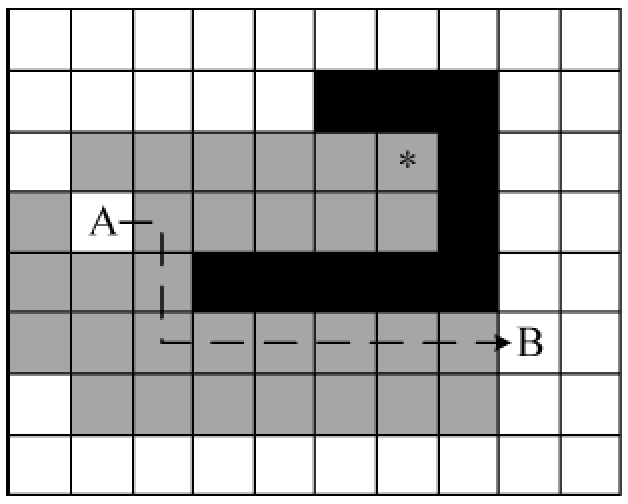


图 4.6　A＊搜索求最短路径

视频讲解

这个方法就是 A＊算法，下面给出它的一般性描述。

在搜索过程中，用一个评估函数对当前情况进行评估，得到最好的状态，从这个状态继续搜索，直到目标。设 x 是当前所在的状态，$f(x)$是对 x 的评估函数，有：

$$f(x)=g(x)+h(x)$$

$g(x)$表示从初始状态到 x 的实际代价，它不体现 x 和终点的关系。

$h(x)$表示 x 到终点的最优路径的评估，它就是“启发式”信息，把 $h(x)$称为启发函数。很显然，$h(x)$决定了 A＊算法的优劣。

特别需要注意的是，$h(x)$不能漏掉最优解。

在上面的例子中，曼哈顿距离就是启发函数 $h(x)$。曼哈顿距离是一种简单而且常用的启发函数。

在上面这个例子中，可以看出 A＊算法包含了 BFS 和贪心算法。

(1) 如果 $h(x)=0$，有 $f(x)=g(x)$，就是普通的 BFS 算法，会访问大量的方块。

(2) 如果 $g(x)=0$，有 $f(x)=h(x)$，就是贪心算法，此时图中标注“＊”的方块也会被访问到。贪心法的缺点是可能陷在局部最优中，例如陷在“＊”的方块中，被堵在障碍后面，无法到达终点。

3. A＊算法与八数码问题

八数码问题也可以用 A＊算法进行优化。通常考虑 3 种估价函数：

(1) 以不在目标位置的数码的个数作为估价函数。

(2) 以不在目标位置的数码与目标位置的曼哈顿距离作为估价函数。

(3) 以逆序数[①]作为估价函数。

第(2)种比第(1)种好，可作为八数码问题的估价函数。

4.3.4　双向广搜

双向广搜是 BFS 的增强版。

① 逆序数可以用来判断八数码是否有解。

前面提到，可以把 BFS 想象成在一个平静的池塘丢一颗石头，激起的波浪一层层扩散到整个空间，直到到达目标，就得到了从起点到目标点的最优路径。那么，如果同时在起点和目标点向对方做 BFS，两个石头激起的波浪向对方扩散，将在中间的某个位置遇到，此时即得到了最优路径。在绝大多数情况下，双向广搜比只做一次 BFS 搜索的空间要少很多，从而更有效率。

从上面的描述可知，双向广搜的应用场合是知道起点和终点，并且正向和逆向都能进行搜索。

下面是一个典型的双向广搜问题。

> **hdu 1401 "Solitaire"**
>
> 有一个 8×8 的棋盘，上面有 4 颗棋子，棋子可以上下左右移动。给定一个初始状态和一个目标状态，问能否在 8 步之内到达。

题目确定了起点和终点，十分适合双向 BFS。要求在 8 步之内到达，可以从起点和终点分别开始，各自广搜 4 步，如果出现交点则说明可达。读者可以练习此题，虽然程序比较烦琐，有很多细节需要处理，但是难度不高。

4.3.2 节讲解的八数码问题也非常适合使用双向广搜技术进行优化。

【习题】

hdu 1401 "Solitaire"；

hdu 3567 "Eight II"，用双向广搜解决八数码问题。

4.4　DFS

4.4.1　DFS 和递归

hdu 1312 题有另外一种解决方案，即 4.3.1 节中提到的"一只老鼠走迷宫"。设 num 是到达的砖块数量，算法过程描述如下：

(1) 在初始位置令 num=1，标记这个位置已经走过。

(2) 左、上、右、下 4 个方向，按顺时针顺序选一个能走的方向，走一步。

(3) 在新的位置 num++，标记这个位置已经走过。

(4) 继续前进，如果无路可走，回退到上一步，换个方向再走。

(5) 继续以上过程，直到结束。

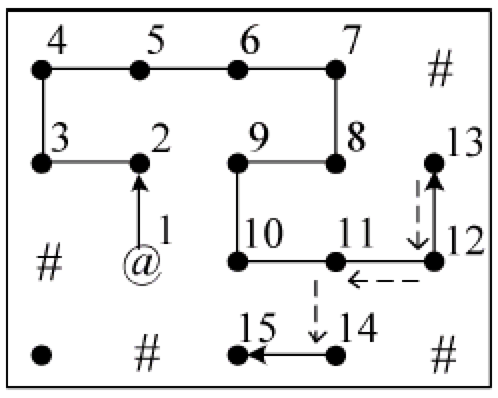


图 4.7　DFS 过程

在以上过程中，能够访问到所有合法的砖块，并且每个砖块只访问一次，不会重复访问(回退不算重复)，如图 4.7 所示。

hdu 1312 的路线如下：从 1 到 13，能一直走下去。在 13 这个位置，到底了不能再走，按顺序回退到 12、11；在 11 这个位置，换个方向又能走到 14、15。到达 15 后，发现不能再走下去，那么按顺

序倒退，即 14→11→10→9→8→7→6→5→4→3→2→1，在这个过程中发现全部都没有新路，最后退回到起点，结束。

为加深对递归的理解，这里再次给出递归返回的完整顺序，即 13→12→15→14→11→10→9→8→7→6→5→4→3→2→1。

在这个过程中，最重要的特点是在一个位置只要有路，就一直走到最深处，直到无路可走，再退回一步，看在上一步的位置能不能换个方向继续往下走。这样就遍历了所有可能走到的位置。

这个思路就是深度搜索。从初始状态出发，下一步可能有多种状态；选其中一个状态深入，到达新的状态；直到无法继续深入，回退到前一步，转移到其他状态，然后再深入下去。最后，遍历完所有可以到达的状态，并得到最终的解。

上述过程用 DFS 实现是最简单的，代码比 BFS 短很多。

下面是代码。读者可以在 DFS()函数中打印走过的位置以及回退的情况。从打印的信息可以看出，在到达 15 后，程序确实是逐步回退到起点的。

```
//用 DFS()替换 4.3.1 节程序中的 BFS(),并在 main()中的相同位置调用它
void DFS(int dx, int dy){
    room[dx][dy] = '#';                //标记这个位置,表示已经走过
     //cout <<"walk:"<< dx << dy << endl; //在此处打印走过的位置,验证是否符合
    num++;
    for(int i = 0; i < 4; i++) {        //左、上、右、下 4 个方向顺时针深搜
        int newx = dx + dir[i][0];
        int newy = dy + dir[i][1];
        if(CHECK(newx, newy) && room[newx][newy] == '.'){
            DFS(newx, newy);
            //cout <<"      back:"<< dx << dy << endl;
            //在此处打印回退的点的坐标,观察深搜到底后回退的情况
            //例如到达最后的 15 这个位置后会一直退到起点
            //即打印出 14-11-10-9-8-7-6-5-4-3-2-1.这也是递归程序返回的过程
        }
    }
}
```

4.4.2 回溯与剪枝

前面提到的 DFS 搜索，基本的操作是将所有子结点全部扩展出来，再选取最新的一个结点进行扩展。

不过，在很多情况下，用递归列举出所有的路径可能会因为数量太大而超时。由于很多子结点是不符合条件的，可以在递归的时候“看到不对头就撤退”，中途停止扩展并返回。这个思路就是回溯，在回溯中用于减少子结点扩展的函数是剪枝函数。

大部分 DFS 搜索题目都需要用到回溯的思路，其难度主要在于扩展子结点的时候如何构造停止递归并返回的条件。这需要通过大量地练习有关题目才能熟练应用。

八皇后问题是经典的回溯与剪枝的应用。

八皇后问题。在棋盘上放置 8 个皇后，使得它们不同行、不同列、不同对角线。N 皇后问题是八皇后问题的扩展。

如果用暴力方法，先排列出所有的棋局，然后一一判断、去除非法的棋局，请读者自己思考复杂度有多大。

下面以四皇后问题为例描述解题过程。在图 4.8 中，从第 1 行开始放皇后：第 1 行从左到右有 4 种方案，产生 4 个子结点；第 2 行，排除同列和斜线，扩展新的子结点，注意不用排除同行，因为第 2 行和第 1 行已经不同行；继续扩展第 3 行和第 4 行，结束。

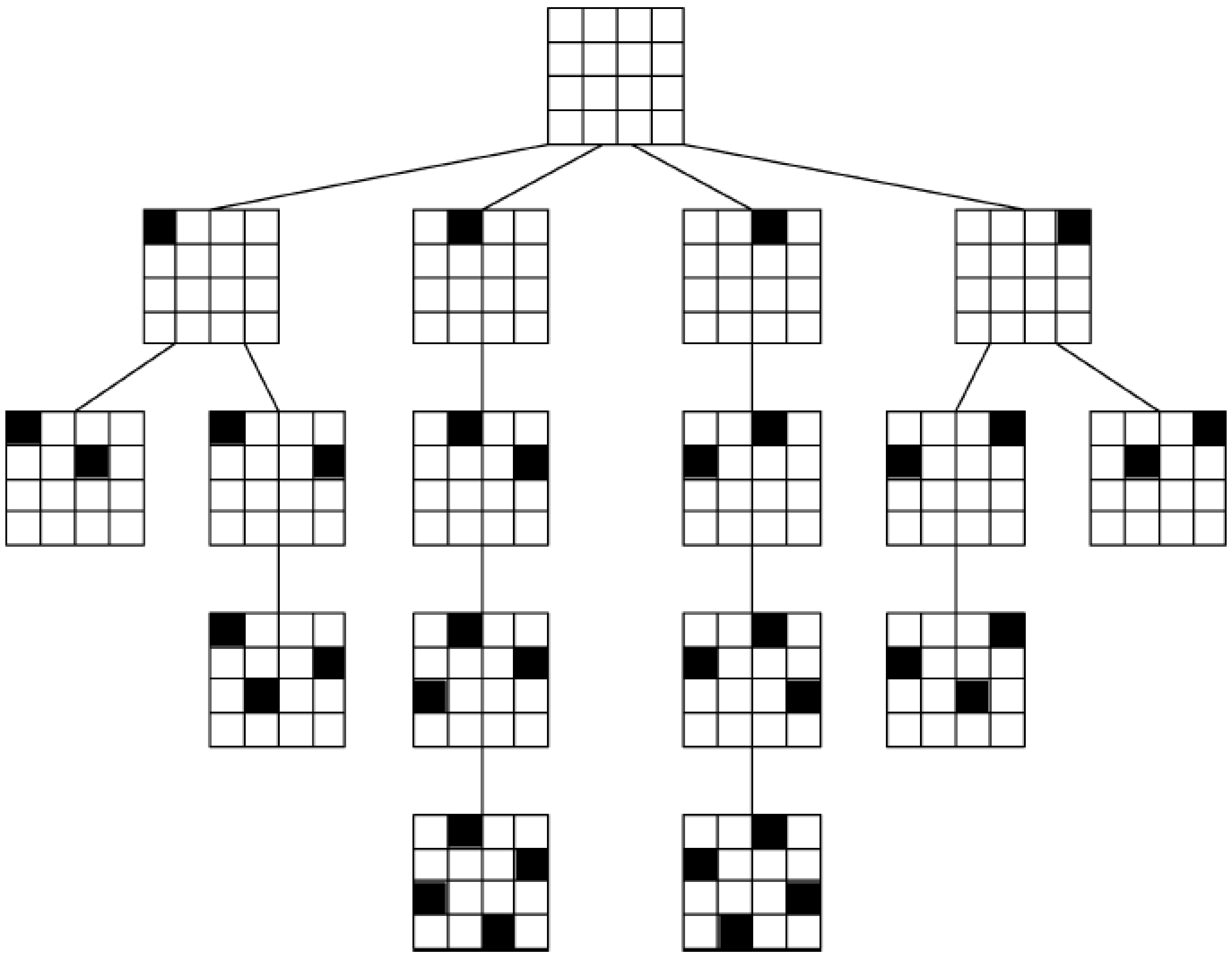


图 4.8 四皇后问题的搜索树

该图用 BFS 和 DFS 都能实现。前文说过，DFS 的代码比 BFS 简洁很多。下面用 DFS 来解决。

关键问题：在扩展结点时如何去掉不符合条件的子结点？

设左上角是原点(0,0)，已经放好的皇后的坐标是(i,j)，不同行、不同列、不同斜线的新皇后的坐标是(c,r)，它们的关系如下：

(1) 横向，不同行：$i \neq r$。

(2) 纵向，不同列：$j \neq c$。

(3) 斜对角：从(i,j)向斜对角走 a 步，那么新坐标(r,c)有 4 种情况，即左上$(i-a,j-a)$、右上$(i+a,j-a)$、左下$(i-a,j+a)$、右下$(i+a,j+a)$，综合起来就是$|i-r|=|j-c|$。新皇后的位置不能放在斜线上，需满足$|i-r| \neq |j-c|$。

下面是 hdu 2553 的代码，求解 N 皇后问题，$N \leqslant 10$。

hdu 2553 "N 皇后问题"

```
#include<bits/stdc++.h>
using namespace std;
int n, tot = 0;
int col[12] = {0};
bool check(int c, int r) {                    //检查是否和已经放好的皇后冲突
    for(int i = 0; i < r; i++)
        if(col[i] == c || (abs(col[i] - c) == abs(i - r)))      //取绝对值
            return false;
```

```
    return true;
}
void  DFS(int r) {                    //一行一行地放皇后,这一次是第 r 行
    if(r == n) {                      //所有皇后都放好了,递归返回
       tot++;                         //统计合法的棋局个数
       return;
    }
    for(int c = 0; c < n; c++)        //在每一列放皇后
        if(check(c, r)){              //检查是否合法
            col[r] = c;               //在第 r 行的 c 列放皇后
            DFS(r + 1);               //继续放下一行皇后
        }
}
int main() {
    int ans[12] = {0};
    for(n = 0; n <= 10; n++){         //算出所有 N 皇后的答案.先打表,不然会超时
        memset(col,0,sizeof(col));    //清空,准备计算下一个 N 皇后问题
        tot = 0;
        DFS(0);
        ans[n] = tot;                 //打表
    }
    while(cin >> n) {
        if(n == 0)
           return 0;
        cout << ans[n] << endl;
    }
    return 0;
}
```

N 皇后问题的 DFS 回溯程序非常简单,关键有两处,一是如何递归,二是如何剪枝和回溯。在上述程序中有很多细节,例如:

(1) 打表。在 main()中提前算出了从 1 到 10 的所有 N 皇后问题的答案,并存储在数组中,等读取输入后立刻输出。如果不打表,而是等输入 N 后再单独计算输出,会超时。

(2) 递归搜索 DFS()。递归程序十分简洁,把第 1 个皇后按行放到棋盘上,然后递归放置其他的皇后,直到放完。

(3) 回溯判断 check()。判断新放置的皇后和已经放好的皇后在横向、纵向、斜对角方向是否冲突。其中,横向并不需要判断,因为在递归的时候已经是按不同的行放置的。

(4) 模块化编程。例如 check()的内容很少,其实可以直接写在 DFS()内部,不用单独写成一个函数。但是单独写成函数,把功能模块化,好处很多,例如逻辑清晰、容易查错等。建议在写程序的时候尽量把能分开的功能单独写成函数,这样可以大大减少编码和调试的时间。

(5) 复杂度。在上述程序中,DFS()一行行地放皇后,复杂度为 $O(N!)$;check()检查冲突,复杂度为 $O(N)$;总复杂度为 $O(N\times N!)$。当 $N=10$ 时,已经到千万数量级。读者可以自己在程序中统计运行次数。经本书作者验证,$N=11$ 时计算了 900 万次,$N=12$ 时计算了 5 千万次。因此,对于 $N>11$ 的 N 皇后问题,需要用新的方法①。

① 用数据结构舞蹈链(Dancing Links)或者位运算可以较快地解决 $N=15$ 的 N 皇后问题。对于更大的 N,例如当 $N=27$ 时,有 2.34×10^{17} 个解。N 皇后问题是一个 NP 完全问题,不存在多项式时间的算法。

【习题】

poj 2531 “Network Saboteur”；
poj 1416 “Shredding Company”；
poj 2676 “Sudoku”；
poj 1129 “Channel Allocation”；
hdu 1175 “连连看”；
hdu 5113 “Black And White”。

4.4.3 迭代加深搜索

有这样一些题目，它们的搜索树很特别：不仅很深，而且很宽；深度可能到无穷，宽度也可能极广。如果直接用 DFS，会陷入递归无法返回；如果直接用 BFS，队列空间会爆炸。

此时可以采用一种结合了 DFS 和 BFS 思想的搜索方法，即迭代加深搜索（Iterative Deepening DFS，IDDFS）。具体的操作方法如下：

（1）先设定搜索深度为 1，用 DFS 搜索到第 1 层即停止。也就是说，用 DFS 搜索一个深度为 1 的搜索树。

（2）如果没有找到答案，再设定深度为 2，用 DFS 搜索前两层即停止。也就是说，用 DFS 搜索一个深度为 2 的搜索树。

（3）继续设定深度为 3、4……逐步扩大 DFS 的搜索深度，直到找到答案。

这个迭代过程，在每一层的广度上采用了 BFS 搜索的思想，在具体编程实现上则是 DFS 的。

一个经典的例子是“埃及分数”。

埃及分数[①]

在古埃及，人们使用单位分数的和（形如 $1/a$ 的，a 是自然数）表示一切有理数。例如 $2/3=1/2+1/6$，但不允许 $2/3=1/3+1/3$，因为加数中有相同的。对于一个分数 a/b，表示方法有很多种，但是哪种最好呢？首先，加数少的比加数多的好，其次，加数个数相同的，最小的分数越大越好。例如：

$$19/45=1/3+1/12+1/180$$
$$19/45=1/3+1/15+1/45$$
$$19/45=1/3+1/18+1/30$$
$$19/45=1/4+1/6+1/180$$
$$19/45=1/5+1/6+1/18$$

最好的是最后一种，因为 $1/18$ 比 $1/180$、$1/45$、$1/30$ 都大。给出 a、b（$0<a<b<1000$），编程计算最好的表达方式。

① http://codevs.cn/problem/1288/。

这一题显然是搜索，可以按图 4.9 建立搜索树。每一层的元素是分子为 1、分母递增的分数；从上往下的一个分支，就是一个这个分支上所有的分数相加的组合；找到合适的组合就退出。解答树的规模很大，深度可能无限，每一层的宽度也可能无限。

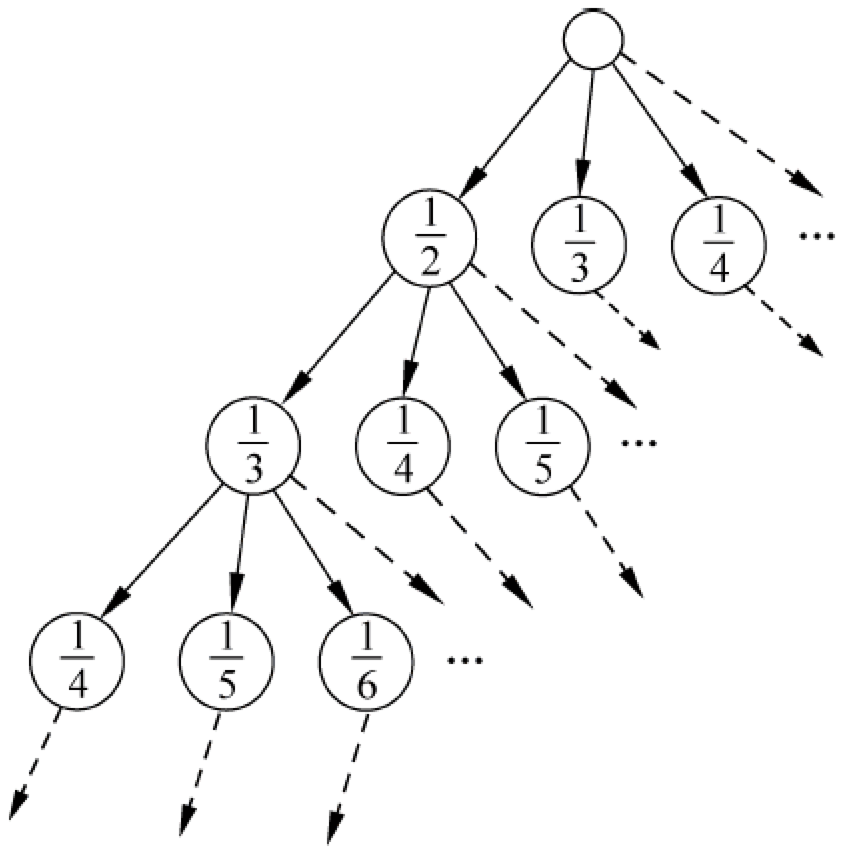


图 4.9　深度和宽度极大的搜索树

在这种情况下适合使用迭代加深搜索。其过程如下：

(1) DFS 到第 1 层，只包括一个分数，如果满足要求就退出。

(2) DFS 前两层，是两个分数的和，例如 1/2+1/3、1/2+1/4、1/2+1/5、…、1/3+1/4、…，找到合适的答案就退出。

(3) DFS 前 3 层……

按上述步骤能搜索到所有可能的组合，并且规避了直接使用 DFS 或 BFS 的弊端。

4.4.4　IDA *

IDA * 是对迭代加深搜索 IDDFS 的优化，可以把 IDA * 看成 A * 算法思想在迭代加深搜索中的应用。

IDDFS 仍然是一种“盲目”的搜索方法，只是把搜索范围约束到了可行的空间内。如果在进行 IDDFS 的时候能预测出当前 DFS 的状态，不再继续深入下去，那么就可以直接返回，不再继续，从而提高了效率。

这个预测就是在 IDDFS 中增加一个估价函数。在某个状态，经过函数计算，发现后续搜索无解，就返回。简单地说，就是在 IDDFS 的过程中利用估价函数进行剪枝操作。

下面这个例题说明了 IDDFS 和估价函数之间的关系。

poj 3134 “Power Calculus”

给定数 x 和 n，求 x^n，只能用乘法和除法，算过的结果可以被利用。问最少算多少次就够了。其中 $n \leqslant 1000$。

这一题等价于从数字 1 开始，用加减法，最少算多少次能得到 n。

搜索的范围是每一步搜索，用前一步得出的值和之前产生的所有值进行加、减运算得到

新的值,判断这个值是否等于 n。

这一题的麻烦在于,每一步搜索,新值的数量增长极快。如果直接用 DFS,深度可能有1000,可能会溢出;如果用 BFS,也可能超出队列范围。

这一题用 IDDFS 非常合适,再用估价函数进行剪枝,可以高效地完成计算。

(1) IDDFS:指定递归深度,每一次做 DFS 时不超过这个深度。

(2) 估价函数:如果当前的值用最快的方式(连续乘 2,倍增)都不能到达 n,停止用这个值继续 DFS。

poj 3134 的代码

```
#include <iostream>
using namespace std;
int val[1010];                               //保存一个搜索路径上每一步的计算结果
int pos, n;
bool ida(int now, int depth){
    if(now > depth)   return false;          //IDDFS:大于当前设定的 DFS 深度,退出
    if(val[pos] << (depth - now) < n)
        return false;                        //估价函数:用最快的倍增都不能到达 n,退出
    if(val[pos] == n)   return true;         //当前结果等于 n,搜索结束
    pos ++ ;
    for(int i = 0 ; i < pos ; i ++) {
        val[pos] = val[pos - 1] + val[i];    //上一个数与前面所有的数相加
        if(ida(now + 1, depth))   return true;
        val[pos] = abs(val[pos - 1] - val[i]); //上一个数与前面所有的数相减
        if(ida(now + 1, depth))   return true;
    }
    pos -- ;
    return false;
}
int main(){
    int t;
    while(cin >> n && n){
        int depth;
        for(depth = 0 ; ; depth ++){         //每次只 DFS 到深度 depth
            val[pos = 0] = 1;                //初始值是 1
            if(ida(0, depth)) break;         //每次都从 0 层开始 DFS 到第 depth 层
        }
        cout << depth << endl;
    }
    return 0;
}
```

【习题】

hdu 1560 “DNA sequence”,经典 IDA * 题目;

hdu 1667 “The Rotation Game”,经典 IDA * 题目。

4.5 小　结

DFS 和 BFS 是算法设计中的基本技术，是基础的基础。

这两种算法都能遍历搜索树的所有结点，区别在于如何扩展下一个结点。DFS 扩展子结点的子结点，搜索路径越来越深，适合采用栈这种数据结构，并用递归算法来实现；BFS 扩展子结点的兄弟结点，搜索路径越来越宽，适合用队列来实现。

1. 复杂度

DFS 和 BFS 对所有的点和边做了一次遍历，即对每个结点均做一次且仅做一次访问。设点的数量是 V，连接点的边总数是 E，那么总复杂度是 $O(V+E)$，看起来复杂度并不高。但是，有些问题的 V 和 E 本身就是指数级的，例如八数码问题的状态，是 $O(n!)$ 的。因此，在搜索时需要用到剪枝、回溯、双向广搜、迭代加深、A＊、IDA＊等方法，尽量减少搜索的范围，使访问的总次数远远小于 $O(V+E)$。

2. 应用场合

DFS 一般用递归实现，代码比 BFS 更短。如果题目能用 DFS 解决，可以优先使用它。

当然，一些问题更适合用 DFS，另一些问题更适合用 BFS。在一般情况下，BFS 是求解最优解的较好方法，例如像迷宫这样的求最短路径问题应该用 BFS，具体内容见第 10 章中的“最短路径”；而 DFS 多用于求可行解。在第 10 章中还有大量应用 BFS 和 DFS 的例子。

第 5 章　高级数据结构

✍ 并查集
✍ 二叉树
✍ Treap 树
✍ Splay 树
✍ 线段树
✍ 树状数组

竞赛题是对输入的数据进行运算，然后输出结果。因此，编写程序的一个基本问题就是数据处理，包括如何存储输入的数据、如何组织程序中的中间数据等。这个技术就是数据结构。学习数据结构，建立计算思维的基础，是成为合格程序员的基本功。本章讲解一些常用的高级数据结构。

数据结构的作用是分析数据、组织数据、存储数据。基本的数据类型有字符和数字，这些数据需要存储在空间中，然后程序按规则读取和处理它们。

数据结构和算法不同，它并不直接解决问题，但是数据结构是算法不可分割的一部分。首先，数据结构把杂乱无章的数据有序地组织起来，逻辑清晰，易于编程处理；其次，数据结构便于算法高效地访问和处理数据，大大减少空间和时间复杂度。

(1) 存储的空间效率。例如一个围棋程序，需要存储棋盘和棋盘上棋子的位置。棋盘可以简单地用一个 19×19 的二维数组（矩阵）表示。每个棋子是一个坐标，例如 $W[5][6]$ 表示位于第 5 行第 6 列的白棋。这种二维数组是数据结构"图"的一种描述，图是描述点和点之间连接关系的数据结构。棋盘只是一种简单的图，更复杂的图例如地图。地图上有两种元素，即点、点之间直连的道路。地图比棋盘复杂，棋盘的每个点只有上、下、左、右 4 个相邻的点，而地图上的一个点可能有很多相邻的点。那么如何存储一个地图？可以简单地用一个二维数组，例如有 n 个点，用一个 $n\times n$ 的二维矩阵表示地图，矩阵上的交叉点 (i,j) 表示第 i 点和第 j 点的连接关系，例如 1 表示相邻，0 表示不相邻。二维矩阵这种数据结构虽然简单、访问速度快，但是用它来存储地图非常浪费空间，因为这是一个稀疏矩阵，其中的交叉点绝大多数等于 0，这些等于 0 的交叉点并不需要存储。一个有 10 万个点的地图，存储它的二维矩阵大小是 100 000×100 000=10GB。所以，在程序中使用二维矩阵来存储地图是不行的，例如手机上的导航软件，常常有几十万个地点，手机存储卡根本放不下。因此，大地图的存储需要用到更有效率的数据结构，这就是邻接表。

(2) 访问的效率。例如输入一大串个数为 n 的无序数字，如果直接存储到一个一维数组里面，那么要查找到某个数据，只能一个个试，需要的时间是 $O(n)$。如果先按大小排序然后再查询，处理起来就很有效率。在 n 个有序的数中找某个数，用折半查找的方法，可以在

$O(\log_2 n)$的时间里找到。

用数据结构存储和处理数据，可以使程序的逻辑更加清晰。

数据结构有以下 3 个要素[1]。

(1) 数据的逻辑结构：线性结构(数组、栈、队列、链表)、非线性结构、集合、图等。

(2) 数据的存储结构：顺序存储(数组)、链式存储、索引存储、散列存储等。

(3) 数据的运算：初始化、判空、统计、查找、遍历、插入、删除、更新等。

常见的数据结构有数组、链表、栈、队列、树、二叉树、集合、哈希、堆与优先队列、并查集、图、线段树、树状数组等。

在第 3 章中已经介绍了基本的数据结构[2]——栈、队列、链表，本章继续讲解一些常用的高级数据结构，包括并查集、二叉树、线段树、树状数组。

5.1 并 查 集

并查集(Disjoint Set)是一种非常精巧而且实用的数据结构，它主要用于处理一些不相交集合的合并问题。经典的例子有连通子图、最小生成树 Kruskal 算法[3]和最近公共祖先(Lowest Common Ancestors，LCA)等。

通常用“帮派”的例子来说明并查集的应用背景。在一个城市中有 n 个人，他们分成不同的帮派；给出一些人的关系，例如 1 号、2 号是朋友，1 号、3 号也是朋友，那么他们都属于一个帮派；在分析完所有的朋友关系之后，问有多少帮派，每人属于哪个帮派。给出的 n 可能是 10^6 的。

读者可以先思考暴力的方法以及复杂度。如果用并查集实现，不仅代码很简单，而且复杂度可以达到 $O(\log_2 n)$。

并查集：将编号分别为 $1\sim n$ 的 n 个对象划分为不相交集合，在每个集合中，选择其中某个元素代表所在集合。在这个集合中，并查集的操作有初始化、合并、查找。

下面先给出并查集操作的简单实现。在这个基础上，后文再进行优化。

视频讲解

1. 并查集操作的简单实现

(1) 初始化。定义数组 int $s[\,]$是以结点 i 为元素的并查集，在开始的时候还没有处理点与点之间的朋友关系，所以每个点属于独立的集，并且以元素 i 的值表示它的集 $s[i]$，例如元素 1 的集 $s[1]=1$。

图 5.1 所示为图解，左边给出了元素与集合的值，右边画出了逻辑关系。为了便于讲解，左边区分了结点 i 和集 s(把集的编号加上了下画线)；右边用圆圈表示集，方块表示元素。

$s[i]$	1	2	3	4	5
i	1	2	3	4	5

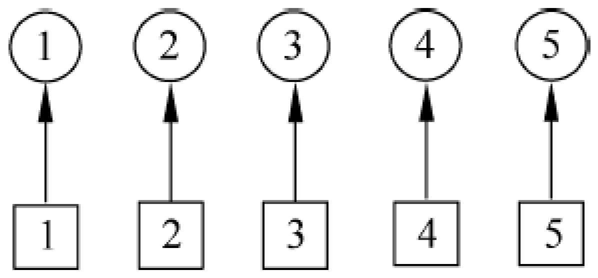


图 5.1 并查集的初始化

① 《数据结构与算法分析新视角》，作者周幸妮等，电子工业出版社。

② 《数据结构(STL 框架)》，作者王晓东，清华大学出版社。

③ 参考本书的“10.10.2 kruskal 算法”。

(2) 合并,例如加入第 1 个朋友关系(1,2),如图 5.2 所示。在并查集 s 中,把结点 1 合并到结点 2,也就是把结点 1 的集1 改成结点 2 的集2。

$s[i]$	*2*	2	3	4	5
i	1	2	3	4	5

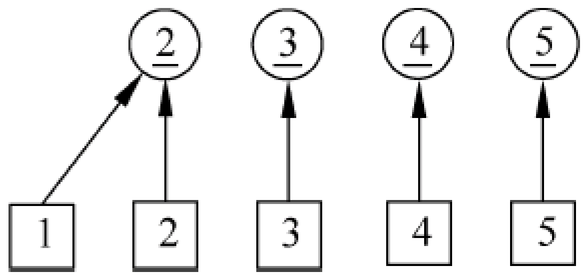


图 5.2　合并(1,2)

(3) 合并,加入第 2 个朋友关系(1,3),如图 5.3 所示。查找结点 1 的集是2,再递归查找元素 2 的集是2,然后把元素 2 的集2 合并到结点 3 的集3。此时,结点 1、2、3 属于一个集。在右图中,为了简化图示,把元素 2 和集2 画在了一起。

$s[i]$	2	*3*	3	4	5
i	1	2	3	4	5

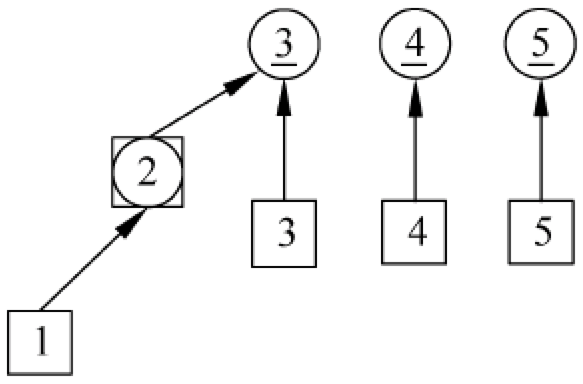


图 5.3　合并(1,3)

(4) 合并,加入第 3 个朋友关系(2,4),如图 5.4 所示。结果如下,请读者自己分析。

$s[i]$	2	3	*4*	4	5
i	1	2	3	4	5

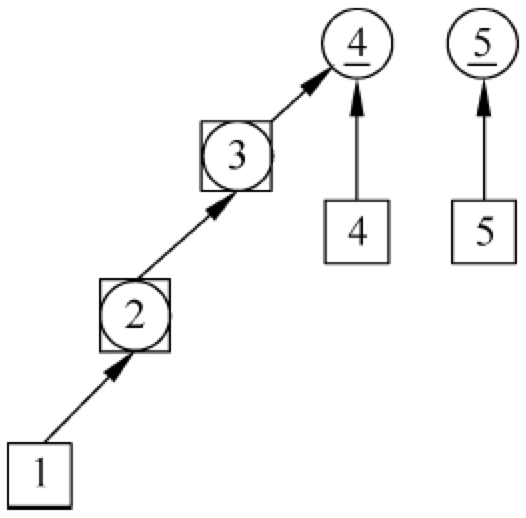


图 5.4　合并(2,4)

(5) 查找。在上面步骤中已经有查找操作。查找元素的集是一个递归的过程,直到元素的值和它的集相等就找到了根结点的集。从上面的图中可以看到,这棵搜索树的高度可能很大,复杂度是 $O(n)$的,变成了一个链表,出现了树的"退化"现象。

(6) 统计有多少个集。如果 $s[i]=i$,这是一个根结点,是它所在的集的代表;统计根结点的数量,就是集的数量。

下面以 hdu 1213 为例实现上述操作。

hdu 1213 "How Many Tables"

有 n 个人一起吃饭,有些人互相认识。认识的人想坐在一起,不想跟陌生人坐。例如 A 认识 B,B 认识 C,那么 A、B、C 会坐在一张桌子上。

给出认识的人,问需要多少张桌子。

一张桌子是一个集,合并朋友关系,然后统计集的数量即可。下面的代码是并查集操作

的具体实现。

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 1050;
int s[maxn];
void init_set(){                        //初始化
   for(int i = 1; i <= maxn; i++)
        s[i] = i;
}
int find_set(int x){                    //查找
    return x == s[x]? x:find_set(s[x]);
}
void union_set(int x, int y){           //合并
    x = find_set(x);
    y = find_set(y);
    if(x != y) s[x] = s[y];
}
int main(){
    int t, n, m, x, y;
    cin >> t;
    while(t--){
        cin >> n >> m;
        init_set();
        for(int i = 1; i <= m; i++){
            cin >> x >> y;
            union_set(x, y);
        }
        int ans = 0;
        for(int i = 1; i <= n; i++)     //统计有多少个集
            if(s[i] == i)
                ans++;
        cout << ans << endl;
    }
    return 0;
}
```

复杂度：在上述程序中，查找 find_set()、合并 union_set()的搜索深度是树的长度，复杂度都是 $O(n)$，性能比较差。下面介绍合并和查询的优化方法，优化之后，查找和合并的复杂度都小于 $O(\log_2 n)$。

2. 合并的优化

在合并元素 x 和 y 时先搜到它们的根结点，然后再合并这两个根结点，即把一个根结点的集改成另一个根结点。这两个根结点的高度不同，如果把高度较小的集合并到较大的集上，能减少树的高度。下面是优化后的代码，在初始化时用 height[i]定义元素 i 的高度，在合并时更改。

```
int height[maxn];
void init_set(){
   for(int i = 1; i <= maxn; i++){
```

```
            s[i] = i;
            height[i] = 0;                         //树的高度
        }
    }
    void union_set(int x, int y){                  //优化合并操作
        x = find_set(x);
        y = find_set(y);
        if (height[x] == height[y]) {
            height[x] = height[x] + 1;             //合并,树的高度加一
            s[y] = x;
        }
        else{                                      //把矮树并到高树上,高树的高度保持不变
            if (height[x] < height[y])  s[x] = y;
            else   s[y] = x;
        }
    }
```

3. 查询的优化——路径压缩

在上面的查询程序 find_set()中,查询元素 i 所属的集需要搜索路径找到根结点,返回的结果是根结点。这条搜索路径可能很长。如果在返回的时候顺便把 i 所属的集改成根结点,如图 5.5 所示,那么下次再搜的时候就能在 $O(1)$的时间内得到结果。

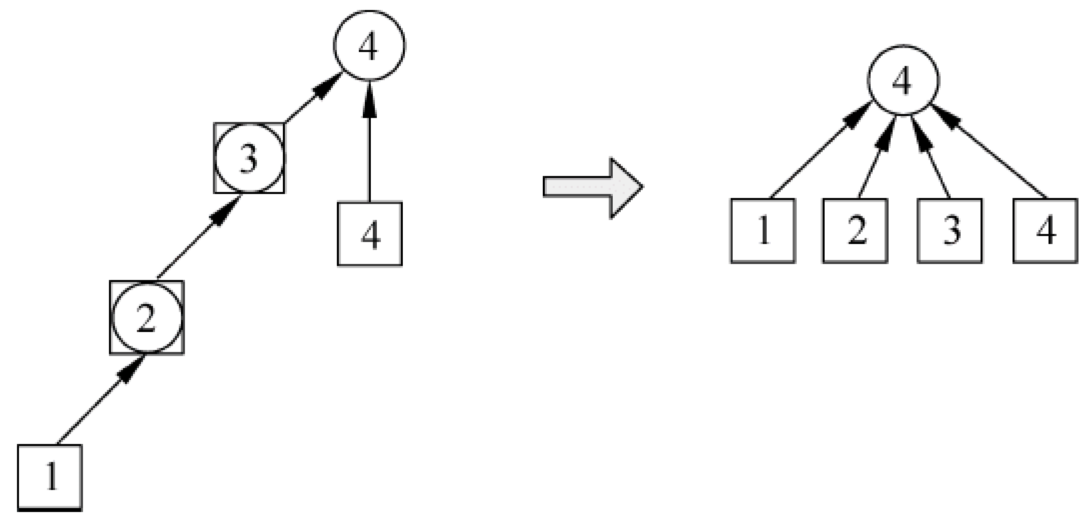


图 5.5 路径压缩

程序如下:

```
int find_set(int x){
    if(x != s[x]) s[x] = find_set(s[x]);          //路径压缩
    return s[x];
}
```

这个方法称为路径压缩,因为在递归过程中,整个搜索路径上的元素(从元素 i 到根结点的所有元素)所属的集都被改为根结点。路径压缩不仅优化了下次查询,而且优化了合并,因为在合并时也用到了查询。

上面的代码用递归实现,如果数据规模太大,担心爆栈,可以用下面的非递归代码:

```
int find_set(int x){
    int r = x;
    while (s[r] != r) r = s[r];                    //找到根结点
    int i = x, j;
    while(i != r){
```

```
            j = s[i];                       //用临时变量 j 记录
            s[i] = r;                       //把路径上元素的集改为根结点
            i = j;
        }
        return r;
    }
```

【习题】

poj 2524 “Ubiquitous Religions”，并查集简单题。

poj 1611 “The Suspects”，简单题。

poj 1703 “Find them, Catch them”。

poj 2236 “Wireless Network”。

poj 2492 “A Bug's Life”。

poj 1988 “Cube Stacking”。

poj 1182 “食物链”，经典题。

hdu 3635 “Dragon Balls”。

hdu 1856 “More is better”。

hdu 1272 “小希的迷宫”。

hdu 1325 “Is It A Tree”。

hdu 1198 “Farm Irrigation”。

hdu 2586 “How far away”，最近公共祖先，并查集＋深搜。

hdu 6109 “数据分割”，并查集＋启发式合并。

5.2 二 叉 树

树是非线性数据结构，它能很好地描述数据的层次关系。树形结构的现实场景很常见，例如，文件目录、书本的目录就是典型的树形结构。

二叉树是最常用的树形结构，特别适合程序设计，常常将一般的树转换成二叉树来处理。本节讲解二叉树的定义、遍历问题，以及二叉搜索树。

5.2.1 二叉树的存储

1. 二叉树的性质

二叉树的每个结点最多有两个子结点，分别是左孩子、右孩子，以它们为根的子树称为左子树、右子树。

二叉树的第 i 层最多有 2^{i-1} 个结点。如果每一层的结点数都是满的，称它为满二叉树。一个 n 层的满二叉树，结点数量一共有 2^n-1 个，可以依次编号为 $1,2,3,\cdots,2^n-1$。如果满二叉树只在最后一层有缺失，并且缺失的编号都在最后，那么称为完全二叉树。满二叉树和完全二叉树图示如图 5.6 所示。

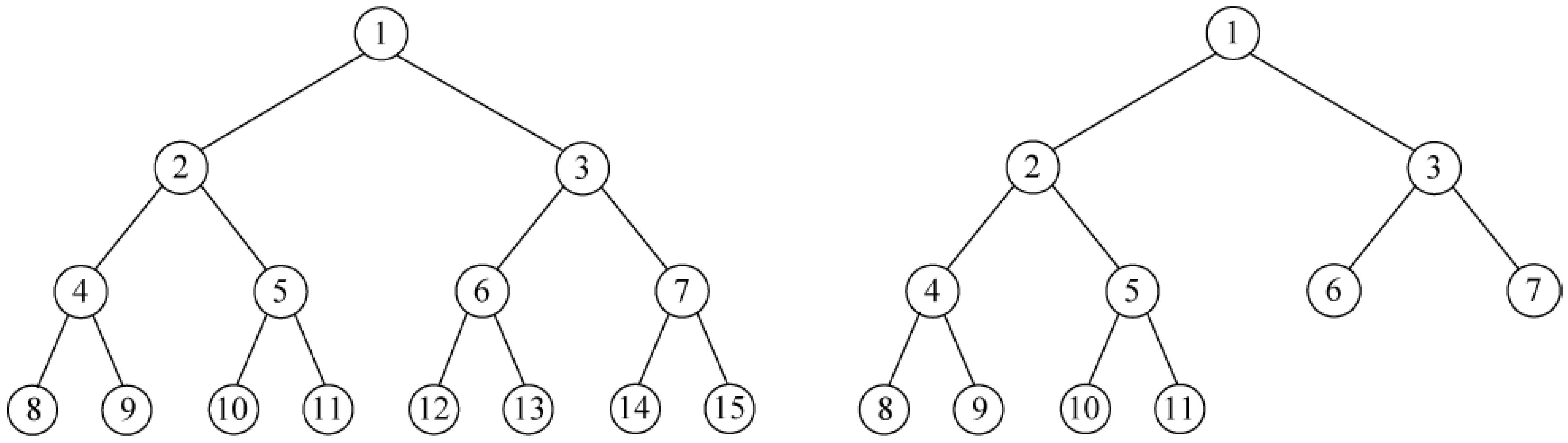


图 5.6　满二叉树和完全二叉树

完全二叉树非常容易操作。一棵结点数量为 k 的完全二叉树，设 1 号点为根结点，有以下性质：

（1）$i>1$ 的结点，其父结点是 $i/2$；

（2）如果 $2i>k$，那么 i 没有孩子；如果 $2i+1>k$，那么 i 没有右孩子；

（3）如果结点 i 有孩子，那么它的左孩子是 $2i$，右孩子是 $2i+1$。

2. 二叉树的存储结构

二叉树一般使用指针来实现，并指向左、右子结点。

```
struct node{
    int value;                          //结点的值
    node *l, *r;                        //指向左、右子结点
};
```

在新建一个 node 时，用 new 运算符动态申请内存。使用完毕后，应该用 delete 释放它，否则会内存泄漏。

二叉树也可以用数组来实现。特别是完全二叉树，用数组来表示父结点和子结点的关系非常简便。

5.2.2　二叉树的遍历

视频讲解

1. 宽度优先遍历

有时需要按层次一层层地遍历二叉树。例如在图 5.7 中需要按 *EBGADFICH* 的顺序访问，那么用宽度优先搜索是最合适的。用队列实现搜索的过程见本书 4.3 节。

2. 深度优先遍历

用深度优先搜索遍历二叉树，代码极其简单。

按深度搜索的顺序访问二叉树，对根（父）结点、左儿子、右儿子进行组合，有先（根）序遍历、中（根）序遍历、后（根）序遍历这 3 种访问顺序，这里默认左儿子在右儿子的前面。

（1）先序遍历。即按父结点、左儿子、右儿子的顺序访问。在图 5.7 中，访问返回的顺序是 *EBADCGFIH*。

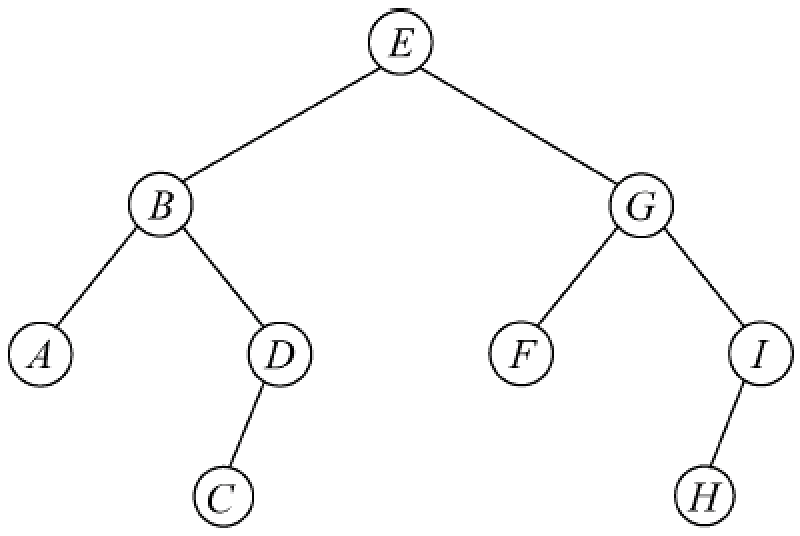


图 5.7　二叉树的遍历

先序遍历的第 1 个结点是根。先序遍历的伪代码如下：

```
void preorder(node * root){
        cout << root -> value;              //输出
        preorder(root -> l);                //递归左子树
        preorder(root -> r);                //递归右子树
}
```

（2）中序遍历。按左儿子、父结点、右儿子的顺序访问。在图 5.7 中，访问返回的顺序是 *ABCDEFGHI*。读者可能注意到“*ABCDEFGHI*”刚好是字典序，这不是巧合，是因为图示的是一个二叉搜索树。在二叉搜索树中，中序遍历实现了排序功能，返回的结果是一个有序排列。中序遍历还有一个特征：如果已知根结点，那么在中序遍历的结果中，排在根结点左边的点都在左子树上，排在根结点右边的点都在右子树上。例如，*E* 是根，*E* 左边的“*ABCD*”在它的左子树上；再如，在子树“*ABCD*”上，*B* 是子树的根，那么“*A*”在它的左子树上，“*CD*”在它的右子树上。

（3）后序遍历。按左儿子、右儿子、父结点的顺序访问。在图 5.7 中，访问返回的顺序是 *ACDBFHIGE*。后序遍历的最后一个结点是根。

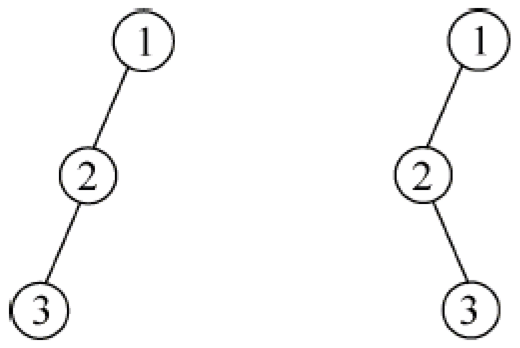


图 5.8　“先序遍历＋后序遍历”不能确定一棵树

如果已知某棵二叉树的 3 种遍历，可以把这棵树构造出来，即“中序遍历＋先序遍历”或者“中序遍历＋后序遍历”，都能确定一棵树。

但是，如果不知道中序遍历，只有“先序遍历＋后序遍历”，不能确定一棵树。例如图 5.8 中两棵不同的二叉树，它们的先序遍历都是“1 2 3”，后序遍历都是“3 2 1”。

上述几种 DFS 遍历的实现见下面例题给出的代码。

> **hdu 1710 “Binary Tree Traversals”**
>
> 输入二叉树的先序和中序遍历序列，求后序遍历。
>
> （1）输入样例。
>
> 先序：1 2 4 7 3 5 8 9 6
>
> 中序：4 7 2 1 8 5 9 3 6
>
> （2）输出样例。
>
> 后序：7 4 2 8 9 5 6 3 1

建树的过程如下：

（1）先序遍历的第 1 个数是整棵树的根，例如样例中的“1”。知道了“1”是根，对照中序遍历，“1”左边的“4 7 2”都在根的左子树上，右边的“8 5 9 3 6”都在根的右子树上。

（2）递归上述过程。例如，上面步骤得到的中序遍历的“4 7 2”，对照先序的第 2 个数是“2”，那么“2”是左子树的根，在中序遍历的“4 7 2”中，“2”左边的“4 7”都在以“2”为根的左子树上，等等。

图 5.9 所示为示意图，画线的数字是读取先序遍历逐一处理的当前步骤的根，方框内是中序遍历的部分数字。

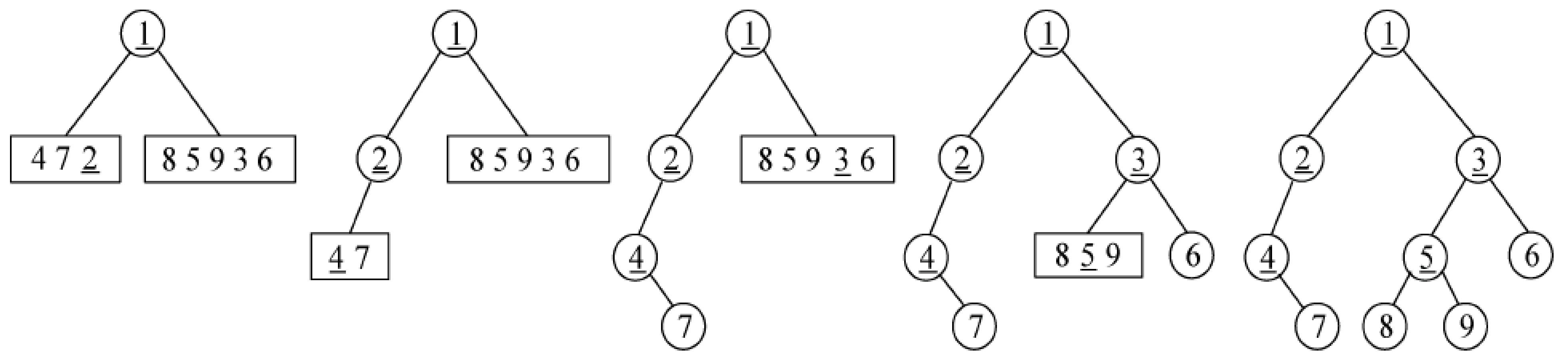


图 5.9　用先序遍历和中序遍历建二叉树

下面是 hdu 1710 的代码，其中 preorder()、inorder()、postorder()分别是先序遍历、中序遍历和后序遍历。可以看到，用 DFS 实现的二叉树遍历代码非常简单。

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1010;
int pre[N], in[N], post[N];                    //先序、中序、后序
int k;
struct node{
    int value;
    node *l, *r;
    node(int value = 0, node *l = NULL, node *r = NULL):value(value), l(l), r(r){}
};
void buildtree(int l, int r, int &t, node* &root) {    //建树
    int flag = -1;
    for(int i = l; i <= r; i++)                //先序的第 1 个数是根,找到对应的中序的位置
        if(in[i] == pre[t]){
            flag = i; break;
        }
    if(flag == -1) return;                     //结束
    root = new node(in[flag]);                 //新建结点
    t++;
    if(flag > l)  buildtree(l, flag - 1, t, root ->l);
    if(flag < r)  buildtree(flag + 1, r, t, root ->r);
}
void preorder(node *root){                     //求先序序列
    if(root != NULL){
        post[k++] = root ->value;              //输出
        preorder(root ->l);
        preorder(root ->r);
    }
}
void inorder(node *root){                      //求中序序列
    if(root != NULL){
        inorder (root ->l);
        post[k++] = root ->value;              //输出
        inorder(root ->r);
    }
}
void postorder(node *root){                    //求后序序列
```

```
        if(root != NULL){
            postorder(root ->l);
            postorder(root ->r);
            post[k++] = root ->value;           //输出
        }
    }
    void remove_tree(node * root){              //释放空间
        if(root == NULL) return;
        remove_tree(root->l);
        remove_tree(root->r);
        delete root;
    }
    int main(){
        int n;
        while(~scanf("%d", &n)){
            for(int i=1;i<=n;i++) scanf("%d", &pre[i]);
            for(int j=1;j<=n;j++) scanf("%d", &in[j]);
            node * root;
            int t = 1;
            buildtree(1, n, t, root);
            k = 0;                              //记录结点个数
            postorder(root);
            for(int i=0;i<k;i++) printf("%d%c",post[i],i==k-1?'\n':' ');
              //作为验证,这里可以用 preorder()和 inorder()检查先序和中序遍历
            remove_tree(root);
        }
        return 0;
    }
```

代码中的 remove_tree()释放申请的空间,如果不释放,会内存泄漏,造成内存浪费。释放空间是标准的、正确的操作。不过,竞赛题目的代码很少,即使不释放空间,也不会出错;而且程序终止后,它申请的空间也会被系统收回。

5.2.5 节给出了用数组实现二叉树的例子。

5.2.3 二叉搜索树

BST(Binary Search Tree,二叉搜索树)是非常有用的数据结构,它的结构精巧、访问高效。BST 的特征如下:

(1) 每个元素有唯一的键值,这些键值能比较大小。通常把键值存放在 BST 的结点上。

(2) 任意一个结点的键值,比它左子树的所有结点的键值大,比它右子树的所有结点的键值小。也就是说,在 BST 上,以任意结点为根结点的一棵子树仍然是 BST。BST 是一棵有序的二叉树。可以推出,键值最大的结点没有右儿子,键值最小的结点没有左儿子。

图 5.10 是一棵二叉搜索树,用中序遍历可以得到它的有序排列。右图的虚线把每个结点隔开,很容易看出,结点正好按从小到大的顺序被虚线隔开了。有虚线的帮助,很容易理解后文介绍 Treap 树和 Splay 树时提到的“旋转”技术。

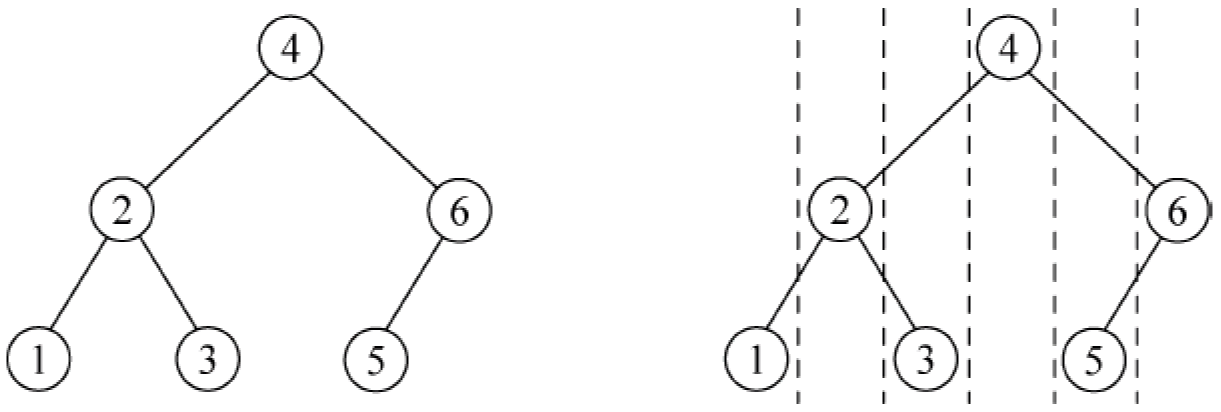


图 5.10　二叉搜索树

数据的基本操作是插入、查询、删除。给定一个数据序列，如何实现 BST？下面给出一种朴素的实现方法。

(1) 建树和插入。以第 1 个数据 x 为根结点，逐个插入其他所有数据。插入过程从根结点开始，如果数据 y 比根结点 x 小，就往 x 的左子树上插，否则就往右子树上插；如果子树为空，就直接放到这个空位，如果非空，就与子树的值进行比较，再进入子树的下一层，直到找到一个空位置。新插入的数据肯定位于一个最底层的叶子结点，而不是插到中间某个结点上替代原来的数据。

从建树的过程可知，如果按给定序列的顺序进行插入，最后建成的 BST 是唯一的。形成的 BST 可能很好，也可能很坏。在最坏的情况下，例如一列有序整数{1, 2, 3, 4, 5, 6, 7}，按顺序插入，会全部插到右子树上；BST 退化成一个只包含右子树的链表，从根结点到最底层的叶子，深度是 n，导致访问一个结点的复杂度是 $O(n)$。在最好的情况下，例如序列{4, 2, 1, 3, 6, 5, 7}，得到的 BST 左、右子树是完全平衡的，深度是 $\log_2 n$，访问复杂度是 $O(\log_2 n)$。退化的 BST 和平衡 BST 如图 5.11 所示。

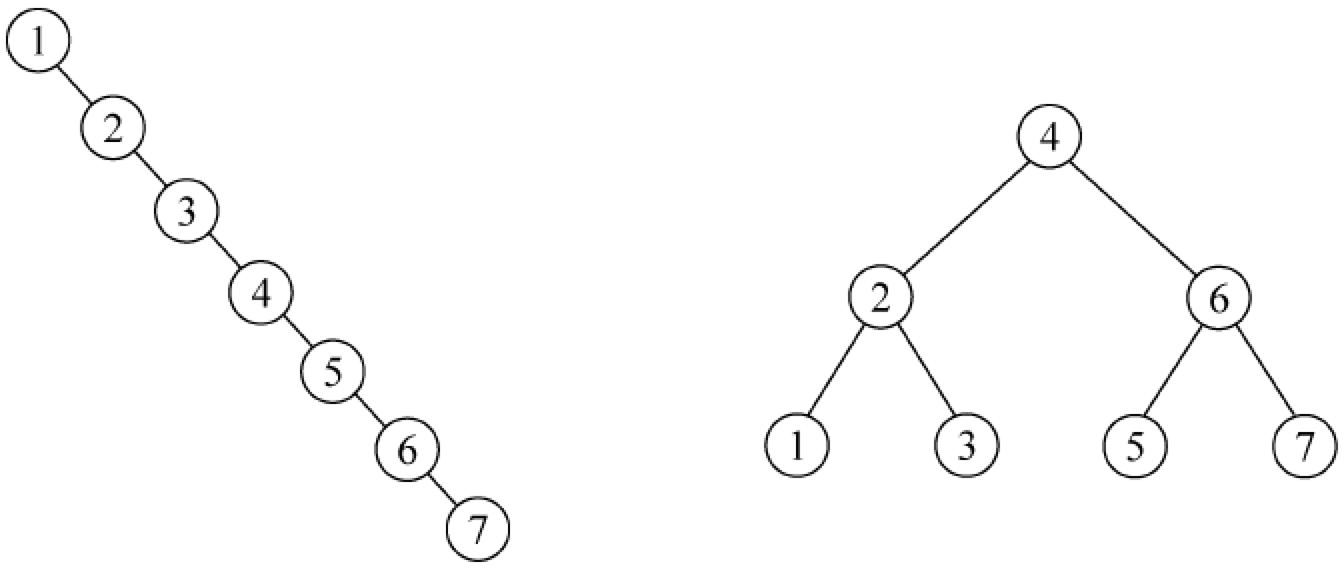


图 5.11　退化的 BST 和平衡 BST

(2) 查询。建树过程实际上也是一个查询过程，所以查询仍然是从根结点开始的递归过程。访问的复杂度取决于 BST 的形态。

(3) 删除。在删除一个结点 x 后，剩下的部分应该仍然是一个 BST。首先找到被删结点 x，如果 x 是最底层的叶子结点，直接删除；如果 x 只有左子树 L 或者只有右子树 R，直接删除 x，原位置由 L 或 R 代替。如果 x 左、右子树都有，情况就复杂了，此时，原来以 x 为根结点的子树需要重新建树。一种做法是，搜索 x 左子树中的最大元素 y，移动到 x 的位置，这相当于原来以 y 为根结点的子树，删除了 y，然后继续对 y 的左子树进行类似的操作，这也是一个递归的过程。删除操作的复杂度也取决于 BST 的形态。

(4) 遍历。在 5.2.2 节中提到用中序遍历 BST，返回的是一个从小到大的排序。

根据上述过程可知，BST 的优劣取决于它是否为一个平衡的二叉树。所以，BST 有

关算法的主要功能是努力使它保持平衡。那么如何实现一个平衡的 BST？由于无法提前安排元素的顺序（如果能一次读入所有元素，也能调整顺序，但是会大费周章，没有必要），所以只能在建树之后通过动态调整使它变得平衡。BST 算法的区别就在于用什么办法调整。

BST 算法有 AVL 树、红黑树、Splay 树、Treap 树、SBT 树等。其中容易编程的有 Splay 树、Treap 树等，也是算法竞赛中容易出的题目，本节后续讲解 Treap 树和 Splay 树。

BST 是一个动态维护的有序数据集，用 DFS 对它进行中序遍历可以高效地输出字典序、查找第 k 大的数等。

STL 与 BST。STL 中的 set 和 map 是用二叉搜索树（红黑树）实现的，检索和更新的复杂度是 $O(\log_2 n)$。如果一个题目需要快速访问集合中的数据，可以用 set 或 map 实现，内容见本书第 3 章。

【习题】

hdu 3999 “The order of a Tree”，模拟 BST 的建树和访问。

hdu 3791 “二叉搜索树”，模拟 BST。

poj 2418 “Hardwood Species”，用 map 快速处理字符串。

5.2.4 Treap 树

首先研究一种比较简单的平衡二叉搜索树——Treap 树。

Treap 是一个合成词，把 Tree 和 Heap 各取一半组合而成。Treap 是树和堆的结合，可以翻译成树堆。

二叉搜索树的每个结点有一个键值，除此之外，Treap 树为每个结点人为添加了一个被称为优先级的权值。对于键值来说，这棵树是排序二叉树；对于优先级来说，这棵树是一个堆。堆的特征是：在这棵树的任意子树上，根结点的优先级最大。

1. Treap 树的唯一性

Treap 树的重要性质：令每个结点的优先级互不相等，那么整棵树的形态是唯一的，和元素的插入顺序没有关系。

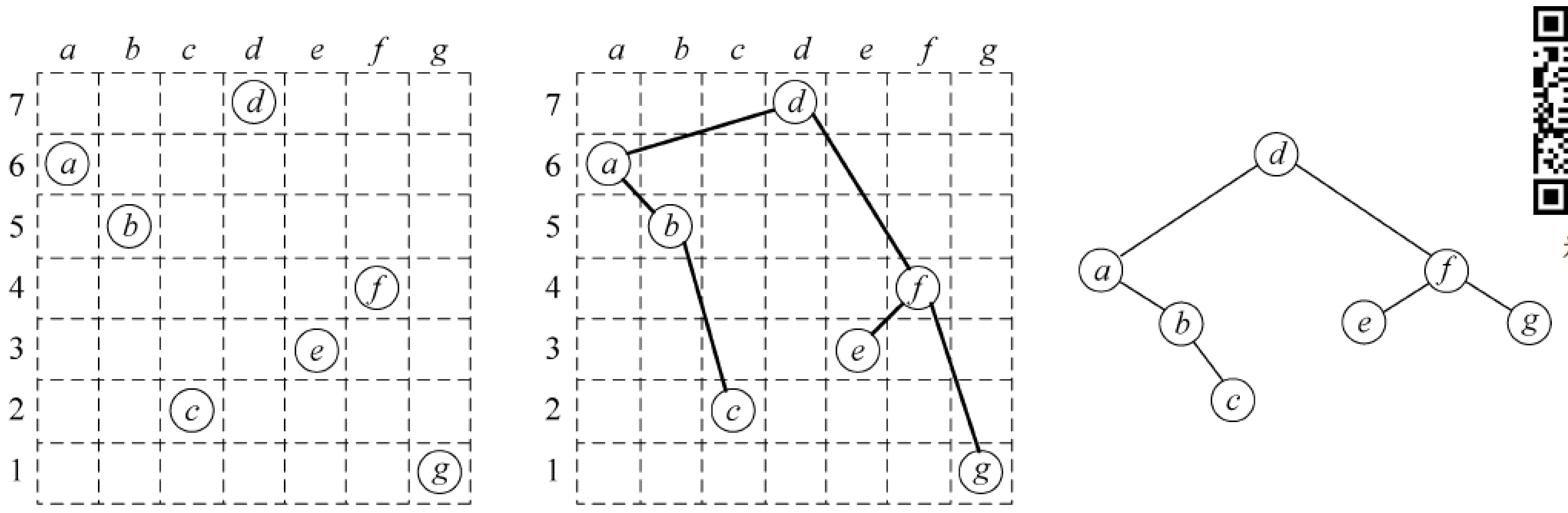
下面用 7 个结点举例说明建树过程，其键值分别是$\{a,b,c,d,e,f,g\}$，优先级分别是$\{6,5,2,7,3,4,1\}$。图 5.12(a)的纵向是优先级，横向是结点的键值；图 5.12(b)按二叉搜索树的规则建了一棵树；图 5.12(c)是结果。从这个图中可以看出 Treap 树的形态是唯一的。

2. Treap 树的平衡问题

从图 5.12 可知，树的形态依赖于结点的优先级。那么如何配置每个结点的优先级，才能避免二叉树的形态退化成链表？最简单的方法是把每个结点的优先级进行随机赋值，那么生成的 Treap 树的形态也是随机的。这虽然不能保证每次生成的 Treap 树一定是平衡的，但是期望[①]的插入、删除、查找的时间复杂度都是 $O(\log_2 n)$的。

① 关于期望的概念，见本书中的“8.4 概率和数学期望”。

视频讲解



(a) 键值和优先级　　(b) 建树　　(c) 形成的Treap树

图 5.12　Treap 树的形态

了解了 Treap 树的概念，读者可以尝试自己完成建树的过程。在阅读下面的内容之前，不妨自己先试一试。

3. Treap 树的插入

如果预先知道所有结点的优先级，那么建树很简单，先按优先级排序，然后按优先级从高到低的顺序插入即可。例如在图 5.12 中，最高优先级的 d 第 1 个插入，是树根；第 2 优先级的 a 比 d 小，插到 d 的左子树上；第 3 优先级的 b 比 a 大，插到 a 的右子树……

不过，其实并不需要这么做。更简单的做法是每读入一个新结点，为它分配一个随机的优先级，插入到树中，在插入时动态调整树的结构，使它仍然是一棵 Treap 树。

把新结点 node 插入到 Treap 树的过程有以下两步：

(1) 用朴素的插入方法把 node 按键值大小插入到合适的子树上。

(2) 给 node 随机分配一个优先级，如果 node 的优先级违反了堆的性质，即它的优先级比父结点高，那么让 node 往上走，替代父结点，最后得到一个新的 Treap 树。

步骤(2)中的调整过程用到了一种技巧——旋转，包括左旋和右旋，如图 5.13 所示。

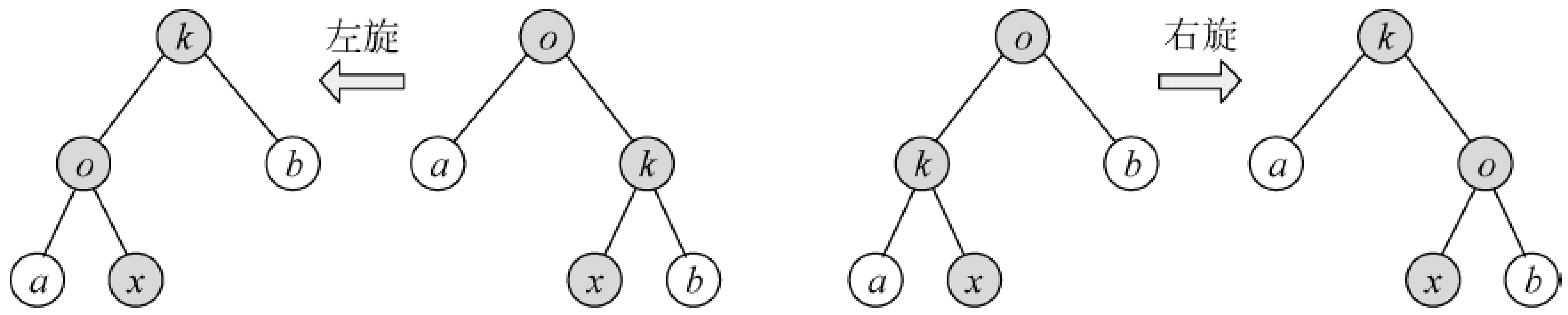


图 5.13　Treap 树的旋转(把 k 旋转到根)

旋转的代码如下，其中 son[0]是左儿子，son[1]是右儿子，代码中定义的结点名称和图 5.13 中的结点名称对应。

```
void rotate(Node * &o, int d){                //d = 0,左旋转; d = 1,右旋
        Node *k = o -> son[d ^1];             //d ^1 与 1 - d 等价,但是更快
        o -> son[d ^1] = k -> son[d];         //图中的 x
        k -> son[d] = o;
        o = k;                                //返回新的根
    }
```

这里仍然以键值为$\{a,b,c,d,e,f,g\}$、优先级为$\{6,5,2,7,3,4,1\}$的 Treap 树为例，调整过程如下：图 5.14(a)是初始 Treap 树；图 5.14(b)插入 d 点，按朴素的插入方法插入到底部；图 5.14(c)中 d 的优先级比父结点 c 高，左旋，上升；图 5.14(d)中 d 的优先级比新的父结点 b 高，继续左旋，上升；图 5.14(e)中，d 再次左旋，上升，完成了新的 Treap 树。

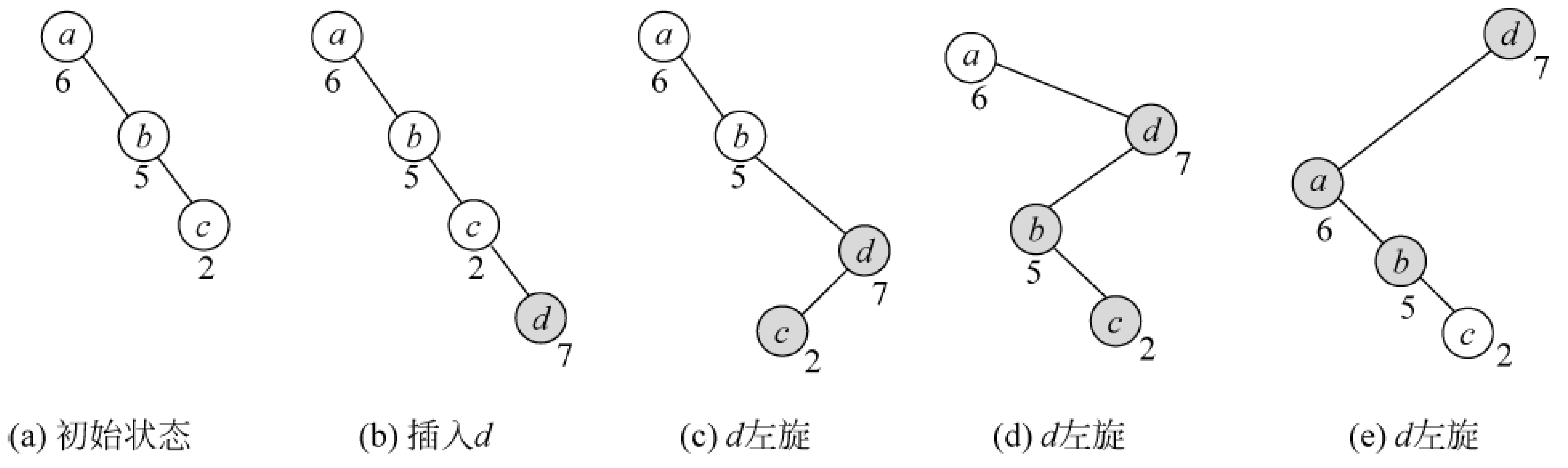


图 5.14　Treap 树的插入和调整

4. Treap 树的删除

如果待删除的结点 x 是叶子结点，直接删除。

如果待删除的结点 x 有两个子结点，那么找到优先级最大的子结点，把 x 向相反的方向旋转，也就是把 x 向树的下层调整，直到 x 被旋转到叶子结点，然后直接删除。

5. 分裂与合并问题

有时需要把一棵树分裂成两棵树，或者把两棵树合并成一棵树。Treap 树做这样的操作是比较烦琐的。读者可以用上面的例子尝试一下分裂和合并，例如在图 5.12(c)中，先把树分成$\{a,b\}$和$\{c,d,e,f,g\}$两棵树，然后再合并。注意在分裂和合并时仍然需要符合 Treap 树的规则。

5.2.5 节提到的 Splay 树做分裂和合并的操作非常简便。

6. Treap 与名次树问题

竞赛中与 Treap 有关的题目很多涉及名次树，例如：

hdu 4585 "Shaolin"

少林寺的第 1 个和尚是方丈，作为功夫大师，他规定每个加入少林寺的年轻和尚要选一个老和尚来一场功夫战斗。每个和尚有一个独立的 id 和独立的战斗等级，新和尚可以选择跟他的战斗等级最接近的老和尚战斗。

方丈的 id 是 1，战斗等级是 10^9。他丢失了战斗记录，不过他记得和尚们加入少林寺的早晚顺序。请帮他恢复战斗记录。

输入：第 1 行是一个整数 n，$0<n\leqslant 100\,000$，和尚的人数，但不包括方丈本人。下面有 n 行，每行有两个整数 k、g，表示一个和尚的 id 和战斗等级，$0\leqslant k,g\leqslant 5\,000\,000$。和尚以升序排序，即按加入少林寺的时间排序。最后一行用 0 表示结束。

输出：按时间顺序给出战斗，打印出每场战斗中新和尚和老和尚的 id。

```
输入样例：
3
2 1
3 3
4 2
0
输出样例：
2 1
3 2
4 2
```

题意很简单，先对老和尚的等级排序，在加入一个新和尚时，找到等级最接近的老和尚，输出老和尚的 id。由于题目给的 n 比较大，因此总复杂度需要是 $O(n\log_2 n)$的。

此题有多种解法，这里给出两种解法——STL map、Treap 树。

1) STL map 代码

STL 的 map 和 set 都是用二叉搜索树实现的。这一题可以用 map 来做。

```
#include <bits/stdc++.h>
using namespace std;
map<int, int> mp;                              //it->first 是等级,it->second 是 id
int main(){
    int n;
    while (~scanf("%d",&n) && n){
        mp.clear();
        mp[1000000000] = 1;                    //方丈 1,等级是 1 000 000 000
        while(n--){
            int id,g;
            scanf("%d%d",&id,&g);              //新和尚 id,等级是 g
            mp[g] = id;                        //新和尚进队
            int ans;
            map<int,int>::iterator it = mp.find(g);    //找到排好序的位置
            if (it == mp.begin())  ans = (++it)->second;
                else{
                map<int,int> :: iterator it2 = it;
                it2--; it++;                   //等级接近的前后两个老和尚
                if (g-it2->first <= it->first-g)
                    ans = it2->second;
                else ans = it->second;
            }
            printf("%d %d\n",id,ans);
        }
    }
    return 0;
}
```

2）Treap 树代码

下面的 Treap 程序[①]给出了 Treap 树的常用操作：定义结点 struct Node、旋转 rotate()、插入 insert()、找第 k 大的数 kth()、查询某个数 find()。

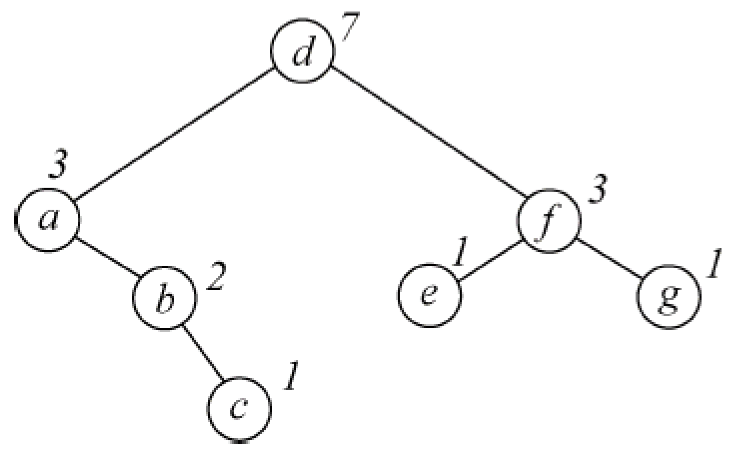


图 5.15　名次树

注意其中的 kth()和 find()，它与名次树问题有关。名次树有两个功能：①找到第 k 大的元素；②查询元素 x 的名次，即 x 排名第几。这两个功能的实现借助于给结点增加的一个 size 值。一个结点的 size 值是以它为根的子树的结点总数量，例如图 5.15 所示的名次树。图中结点上标注的数字就是这个结点的 size。

下面的代码中给出了找第 k 大数的函数 kth()以及查询元素名次的函数 find()，它们的复杂度都是 $O(\log_2 n)$的。

hdu 4585 的 Treap 代码(名次树)

```
#include <bits/stdc++.h>
using namespace std;
int id[5000000 + 5];
struct Node{
    int size;                          //以这个结点为根的子树的结点总数量,用于名次树
    int rank;                          //优先级
    int key;                           //键值
    Node * son[2];                     //son[0]是左儿子, son[1]是右儿子
    bool operator < (const Node &a)const{return rank < a.rank;}
    int cmp(int x)const{
        if(x == key) return -1;
        return x < key?0:1;
    }
    void update(){                     //更新 size
        size = 1;
        if(son[0]!= NULL) size +=  son[0] -> size;
        if(son[1]!= NULL) size +=  son[1] -> size;
        }
};
void rotate(Node * &o, int d){        //d = 0,左旋; d = 1,右旋
        Node * k = o -> son[d^1];      //d^1 与 1 - d 等价,但是更快
        o -> son[d^1] = k -> son[d];
        k -> son[d] = o;
        o -> update();
        k -> update();
        o = k;
    }
void insert(Node * &o, int x){        //把 x 插入到树中
        if(o == NULL){
            o = new Node();
            o -> son[0] = o -> son[1] = NULL;
            o -> rank = rand();
            o -> key = x;
```

① 部分代码改编自《算法竞赛入门经典训练指南》，作者刘汝佳、陈锋，清华大学出版社，3.5.2 节，231 页。

```
            o->size=1;
        }
        else {
            int d=o->cmp(x);
            insert(o->son[d],x);
            o->update();
            if(o<o->son[d])
                rotate(o,d^1);
        }
    }
int kth(Node* o,int k){                     //返回第k大的数
        if(o==NULL||k<=0||k>o->size)
            return -1;
        int s=o->son[1]==NULL?0:o->son[1]->size;
        if(k==s+1) return o->key;
        else if(k<=s) return kth(o->son[1],k);
        else return kth(o->son[0],k-s-1);
    }
int find(Node* o,int k){  //返回元素k的名次
        if(o==NULL)
            return -1;
        int d=o->cmp(k);
        if(d==-1)
            return o->son[1]==NULL? 1: o->son[1]->size+1;
        else if(d==1) return find(o->son[d],k);
        else{
            int tmp=find(o->son[d],k);
            if(tmp==-1) return -1;
            else
                return o->son[1]==NULL? tmp+1 : tmp+1+o->son[1]->size;
        }
    }
int main(){
    int n;
    while(~scanf("%d",&n)&&n){
        srand(time(NULL));
        int k,g;
        scanf("%d%d",&k,&g);
        Node *root=new Node();
        root->son[0]=root->son[1]=NULL;
        root->rank=rand(); root->key=g; root->size=1;
        id[g]=k;
        printf("%d %d\n",k,1);
        for(int i=2;i<=n;i++){
            scanf("%d%d",&k,&g);
            id[g]=k;
            insert(root,g);
            int t=find(root,g);          //返回新和尚的名次
            int ans1,ans2,ans;
            ans1=kth(root,t-1);          //前一名的老和尚
            ans2=kth(root,t+1);          //后一名的老和尚
            if(ans1!=-1&&ans2!=-1)
                ans=ans1-g>=g-ans2 ? ans2:ans1;
            else if(ans1==-1) ans=ans2;
```

```
                else ans = ans1;
                printf("%d %d\n",k,id[ans]);
            }
        }
    return 0;
}
```

【习题】

poj 1442,名次树问题。

hdu 3726 "Graph and Queries",离线算法+Treap 维护名次树。该题非常经典,是必做题。

5.2.5 Splay 树

Splay 树是一种 BST 树,它的查找、插入、删除、分割、合并等操作,复杂度都是 $O(\log_2 n)$ 的。它最大的特点是可以把某个结点往上旋转到指定位置,特别是可以旋转到根的位置,成为新的根结点。它有这样一种应用背景:如果需要经常查询和使用一个数,那么把它旋转到根结点,这样下次访问它,只需要查一次就找到了。

Splay 树有 Treap 树不具备的特点:①Splay 树允许把任意结点旋转到根,而 Treap 树不能,因为它的形态是固定的;②当需要分裂和合并时,Splay 树的操作非常简便。

下面介绍 Splay 操作,其中提根操作是核心。

1. 把结点旋转到根(提根)

Splay 树比 Treap 树的旋转操作的情况更多。

那么如何把一个结点 x 自底向上旋转到根?根据 x 的位置,有以下 3 种情况。

(1) x 的父结点就是根,只需要旋转一次。图 5.16 给出了 x 是根 c 的左儿子的情况,右儿子的情况与之类似。注意观察图中的中序遍历,即二叉搜索树的顺序"$a\ x\ b\ c\ d$",保持不变。

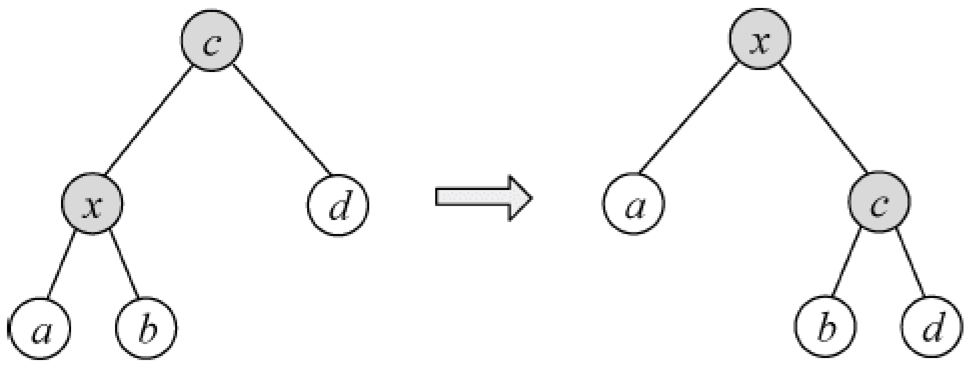


图 5.16 Splay 旋转情况 1

(2) x 的父结点不是根,x、x 的父结点、x 的父结点的父结点,三点共线。此时可以做两次单旋,即先旋转 x 的父结点,再旋转 x,如图 5.17 所示。

(3) x、x 的父结点、x 的父结点的父结点,三点不共线。把 x 按不同方向旋转两次,如图 5.18 所示。

按上述方法可以把任何深度的结点 x 旋转到根。

旋转一次的时间是个常数,那么把 x 从所在的深度提到根,总复杂度是多少?如果是平衡二叉树,最深的结点深度是 $O(\log_2 n)$,那么总复杂度就是 $O(\log_2 n)$。当然二叉树不一

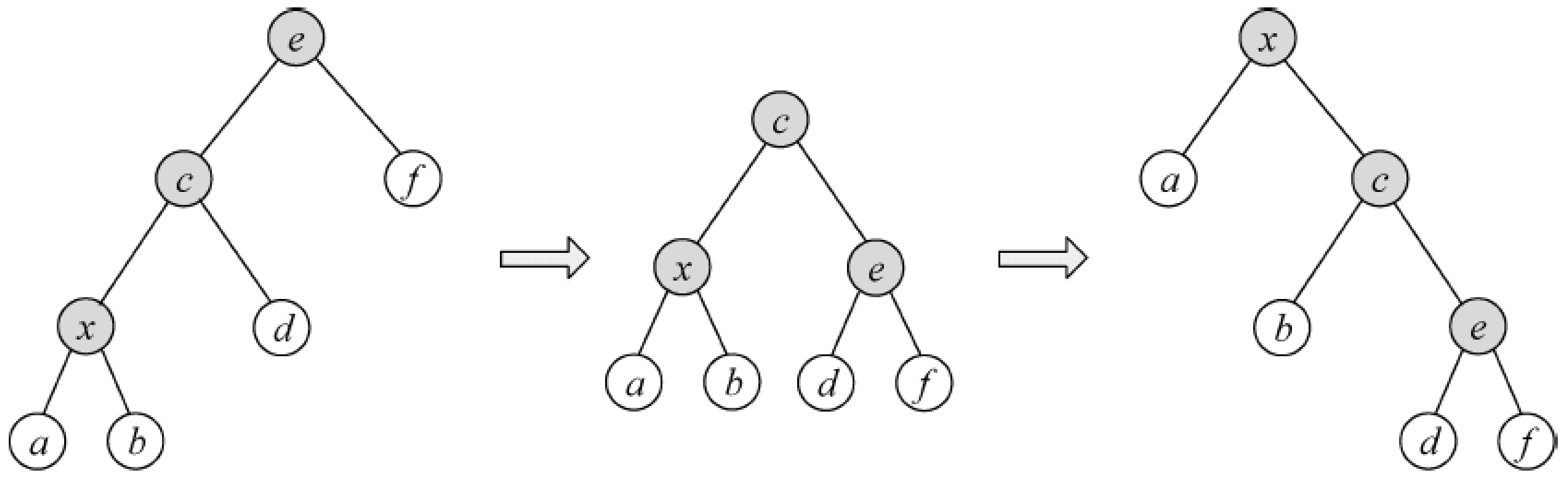


图 5.17　Splay 旋转情况 2

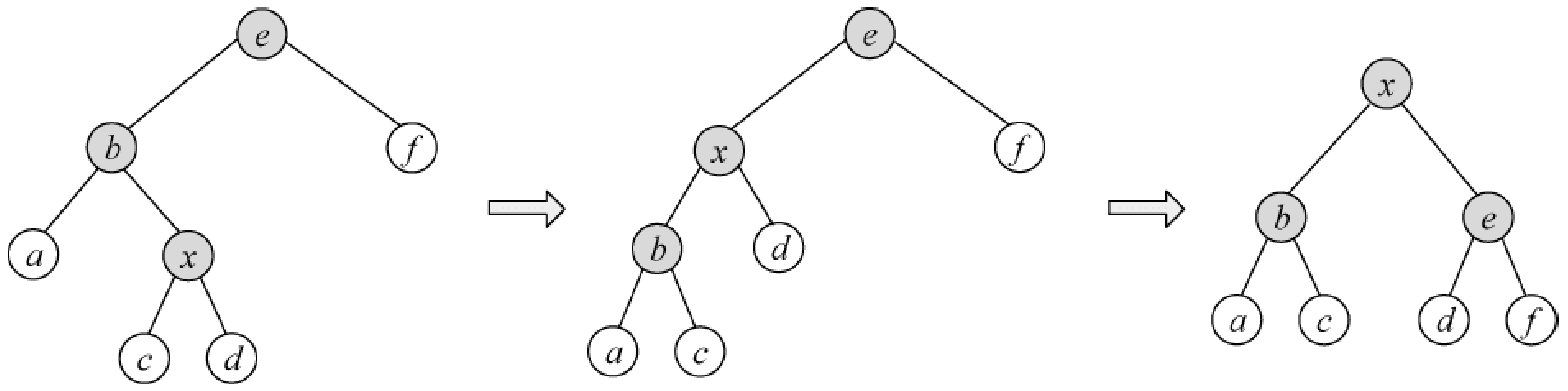


图 5.18　Splay 旋转情况 3

定是平衡的，不过在均摊意义上，可以把 Splay 提根操作的复杂度看成是 $O(\log_2 n)$的。这就是二叉树这种数据结构带来的优势。

下面的插入、分裂、合并，复杂度和提根的复杂度类似。

2. 插入

插入和普通二叉搜索树的方法一样。在插入之后，可以根据需要对新插入的结点做 Splay 操作。

3. 分裂

以第 k 小的数为界，把树分成两部分。先把第 k 小的元素旋转到树根，然后把它与右子树断开，就得到了两棵树。

4. 合并

可合并的两棵树，其中一棵树(设为 left)的所有元素应该小于另一棵树(设为 right)的所有元素。合并过程是先把 left 的最大元素 x 伸展到树根，此时树根 x 没有右子树，把 x 的右子树接到 right 的根，就完成了合并。

5. 删除

把待删除的结点旋转到根，删除它，然后合并左、右子树。

下面的例题给出了 Splay 树的编程细节。

hdu 1890 "Robotic Sort"

有 n 个数字，$1 \leqslant n \leqslant 100\ 000$，用一个机械臂帮忙排序，其方法如图 5.19 所示。在左图中(图中的高度是数字大小)，用机械臂夹住第 1 个数和最小的数，翻转，变成中图的样子，最小的数就处于第 1 个位置。然后对中图用同样的方法找第 2 小的数。继续这个过程直到结束。

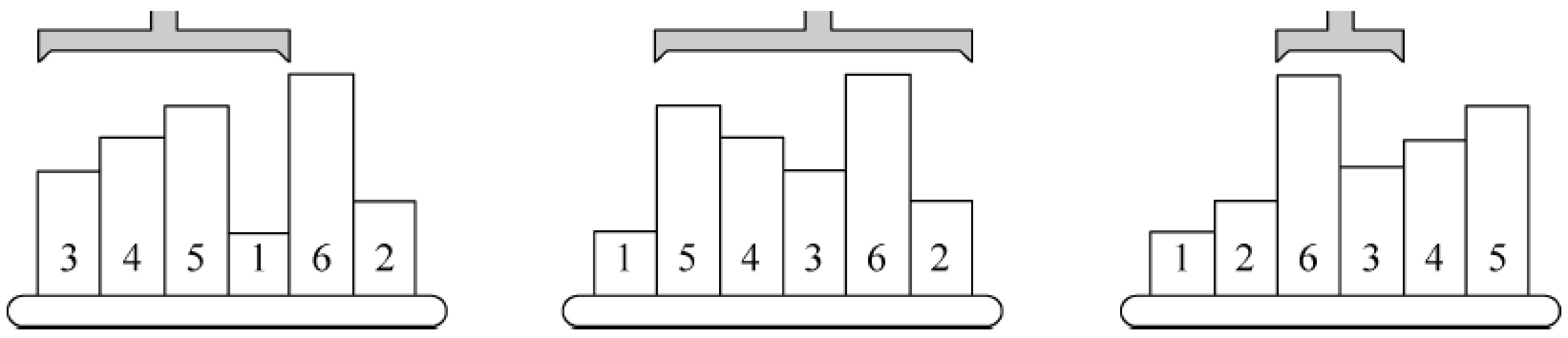


图 5.19　排序方法

输入一些数字，输出第 i 次翻转之前第 i 大的数的位置。

输入样例：3 4 5 1 6 2

输出样例：4 6 4 5 6 6

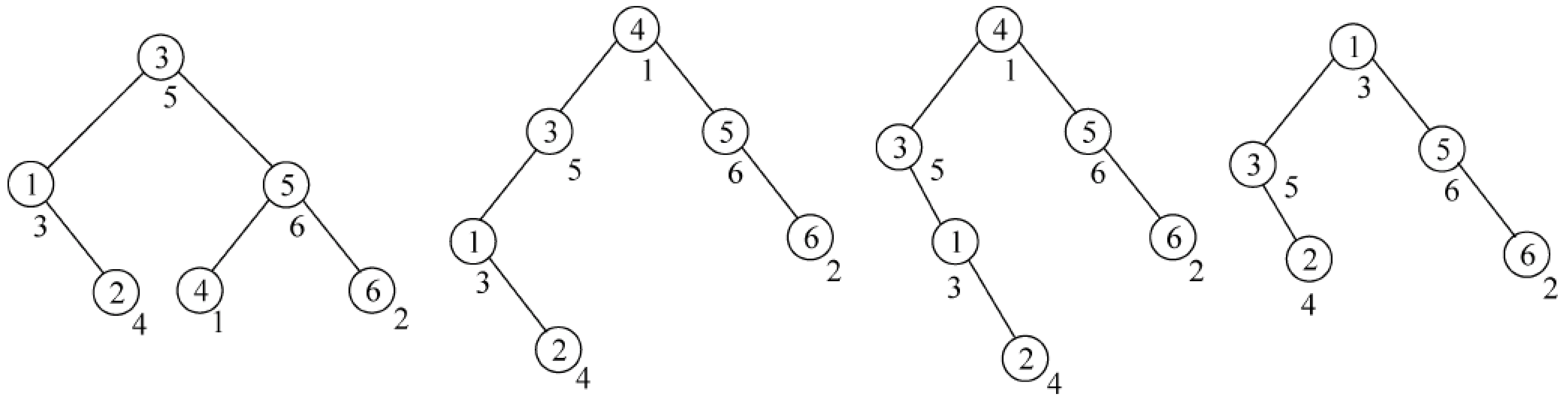
题目的基本操作是找到第 i 大的数，翻转它左边的数(不包括已经处理过的比它小的数)，右边的数保持不变。如果用模拟法编程，复杂度约为 $O(n^3)$，会 TLE。本题需要 $O(n\log_2 n)$的方法。

注意，翻转有两种方法，这里以第 1 次翻转 3 4 5 1 为例：方法 1，直接翻转 3 4 5 1；方法 2，先把 1 挪到最左边，然后翻转 3 4 5。这两种方法的结果一样，在下面的 Splay 程序中适合用第 2 种方法。

本题的操作可以用 Splay 来模拟，利用了 Splay 树能把结点旋转到根的功能。

下面以第 1 个数的处理为例来说明过程。

(1) 建树。把这个序列按初始位置建一个二叉搜索树。图 5.20(a)是建树的结果，圆圈内是初始位置，圆圈旁边的数字是题目给出的序列。根据中序遍历，它是题目的样例3 4 5 1 6 2。建树的代码是 buildtree()。



(a) 建树　(b) 旋转到根　(c) 处理左子树的翻转　(d) 删除根

图 5.20　hdu 1890 题

(2) 用 Splay 旋转到根。找到最小的数，用 Splay 把它旋转到根。其左子树的大小就是数列中排在它左边的数的个数，也就是题目的输出。其右边的数的顺序保持不变，左边的数需要模拟机械臂的翻转。旋转的代码是 splay()。

(3) 翻转左子树。模拟题目中的机械臂翻转，但是，如果每次都完全翻转左子树，时间必然超时。这里从线段树①得到启发，用标记的方式记录翻转情况，减少直接操作的次数，

① 类似线段树的 lazy 操作，见本书中的“5.3.4 区间修改”。

等 Splay 操作的时候再处理。图(c)中只翻转了结点 3,对结点 3 做标记,而它的子树 1、2 保持不变。标记的代码是 update_rev()。注意,翻转会改变 BST 树的有序结构,所以本题并不是 Splay 树的裸题,只是用到了 Splay 树的旋转功能。在下面给出的代码中,如果去掉 update_rev(),就是纯粹的 Splay 代码。

(4) 删除根,即在树上删除最小数。在删除过程中,根据标记进行子树的翻转。最后的结果见图(d),这是去掉了最小数的树,第 1 次处理结束。删除根的代码是 del_root()。

下面是 hdu 1890 的代码。该代码中去掉 update_rev()、pushup()、pushdown(),就是纯的 Splay 代码。

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 100010;
int root;                                   //根
int rev[maxn],pre[maxn],size[maxn];
    //rev[i],标记 i 被翻转;pre[i],i 的父结点;size[i],i 的子树上结点的个数
int tree[maxn][2]; //记录树:tree[i][0],i 的左儿子;tree[i][1],i 的右儿子
struct node{
    int val,id;
    bool operator<(const node &A)const {   //用于 sort()排序
        if(val == A.val)return id < A.id;
        return val < A.val;
    }
}nodes[maxn];
void pushup(int x){                         //计算以 x 为根的子树包含多少子结点
    size[x] = size[tree[x][0]] + size[tree[x][1]] + 1;
}
void update_rev(int x){
    if(!x)return;
    swap(tree[x][0],tree[x][1]);            //翻转 x:交换左右儿子
    rev[x]^ = 1;                            //标记 x 被翻转
}
void pushdown(int x){                       //在做 Splay 时,根据本题的需要,处理机械臂翻转
    if(rev[x]){
        update_rev(tree[x][0]);
        update_rev(tree[x][1]);
        rev[x] = 0;
    }
}
void Rotate(int x,int c){                   //旋转,c = 0 为左旋,c = 1 为右旋
    int y = pre[x];
    pushdown(y);
    pushdown(x);
    tree[y][!c] = tree[x][c];
    pre[tree[x][c]] = y;
    if(pre[y])
        tree[pre[y]][tree[pre[y]][1] == y] = x;
    pre[x] = pre[y];
    tree[x][c] = y;
```

```
        pre[y] = x;
        pushup(y);
    }
    void splay(int x,int goal){
              //把结点 x 旋转为 goal 的儿子,如果 goal 是 0,则旋转到根
        pushdown(x);
        while(pre[x]!= goal){                //一直旋转,直到 x 成为 goal 的儿子
            if(pre[pre[x]] == goal){         //情况(1):x 的父结点是根,单旋一次即可
                pushdown(pre[x]); pushdown(x);
                Rotate(x,tree[pre[x]][0] == x);
            }
            else{                            //x 的父结点不是根
                pushdown(pre[pre[x]]); pushdown(pre[x]); pushdown(x);
                int y = pre[x];
                int c = (tree[pre[y]][0] == y);
                if(tree[y][c] == x){         //情况(2):x、x 的父、x 父的父,不共线
                    Rotate(x,!c);
                    Rotate(x,c);
                }
                else{                        //情况(3):x、x 的父、x 父的父,共线
                    Rotate(y,c);
                    Rotate(x,c);
                }
            }
        }
        pushup(x);
        if(goal == 0) root = x;              //如果 goal 是 0,则将根结点更新为 x
    }
    int get_max(int x){
        pushdown(x);
        while(tree[x][1]){
            x = tree[x][1];
            pushdown(x);
        }
        return x;
    }
    void del_root(){                         //删除根结点
        if(tree[root][0] == 0){
            root = tree[root][1];
            pre[root] = 0;
        }
        else{
            int m = get_max(tree[root][0]);
            splay(m,root);
            tree[m][1] = tree[root][1];
            pre[tree[root][1]] = m;
            root = m;
            pre[root] = 0;
            pushup(root);
        }
    }
```

```
void newnode(int &x, int fa, int val){
    x = val;
    pre[x] = fa;
    size[x] = 1;
    rev[x] = 0;
    tree[x][0] = tree[x][1] = 0;
}
void buildtree(int &x, int l, int r, int fa){ //建树
    if(l > r) return;
    int mid = (l + r)>> 1;
    newnode(x, fa, mid);
    buildtree(tree[x][0], l, mid - 1, x);
    buildtree(tree[x][1], mid + 1, r, x);
    pushup(x);
}
void init(int n){
    root == 0;
    tree[root][0] = tree[root][1] = pre[root] = size[root] = 0;
    buildtree(root, 1, n, 0);
}
int main(){
    int n;
    while(~scanf("%d", &n) && n){
        init(n);
        for(int i = 1; i <= n; i++){
            scanf("%d", &nodes[i].val);  nodes[i].id = i;
        }
        sort(nodes + 1, nodes + n + 1);
        for(int i = 1; i < n; i++){
            splay(nodes[i].id, 0);          //第 i 次翻转:把第 i 大的数旋到根
            update_rev(tree[root][0]);  //左子树需要翻转
            printf("%d ", i + size[tree[root][0]]);
              //i:第 i 次翻转;size:第 i 个被翻转数的左边的个数,就是它左子树的个数
            del_root();                     //删除第 i 次翻转的数,准备下一次翻转
        }
        printf("%d\n", n);
    }
    return 0;
}
```

读者可以在上面代码的基础上写出 Splay 树常见操作的代码,例如:

(1) 查找 x。执行 splay(x, 0),即把 x 旋转到根结点。

(2) 删除 x。先执行 splay(x, 0),把 x 旋转到根,然后用 del_root()删除它。

(3) 查找最大、最小、第 k 大的数。用中序遍历进行查找,查找后可以用 splay()把它旋转到根。

【习题】

hdu 1622,建二叉树;

hdu 3999,二叉树遍历;

hdu 3791,BST;

hdu 4453,Splay 基本题;

hdu 3726,离线处理+Splay,经典题。该题用 Treap 树也能做。

5.3 线 段 树

有这样一类 RMQ(Range Minimum/Maximum Query)问题,求区间最大值或最小值。设有长度为 n 的数列 $\{a_1,a_2,\cdots,a_n\}$,需要进行以下操作。

(1) 求最值:给定 $i,j\leqslant n$,求 $\{a_i,\cdots,a_j\}$ 区间内的最值。

(2) 修改元素:给定 k 和 x,把 a_k 改成 x。

如果用普通数组存储数列,上面两个操作中,求最值的复杂度是 $O(n)$,修改是 $O(1)$。如果有 m 次“修改元素+查询最值”,那么总复杂度是 $O(mn)$。如果 m 和 n 比较大,例如 100 000 以上,那么整个程序的复杂度是 10^{10} 的数量级。这个复杂度在竞赛中是不可承受的。

除了 RMQ 问题以外,类似的还有求区间和问题。对于数列 $\{a_1,a_2,\cdots,a_n\}$,先更改某些数的值,然后给定 $i,j\leqslant n$,求 $\text{sum}=a_i+\cdots+a_j$ 的区间和。对于单个更改或者求和,很容易写出 $O(n)$ 的算法;如果更改和询问的操作总次数是 m,那么整个程序的复杂度是 $O(mn)$。和 RMQ 一样,这样的复杂度也是不行的。

对于这类问题,有一种神奇的数据结构,能在 $O(m\log_2 n)$ 的时间内解决,这就是线段树。

5.3.1 线段树的概念

视频讲解

线段树是一种用于区间处理的数据结构,用二叉树来构造。

线段树是建立在线段(或者区间)基础上的树,树的每个结点代表一条线段[L,R]。图 5.21 所示为是线段[1,5]的线段树。

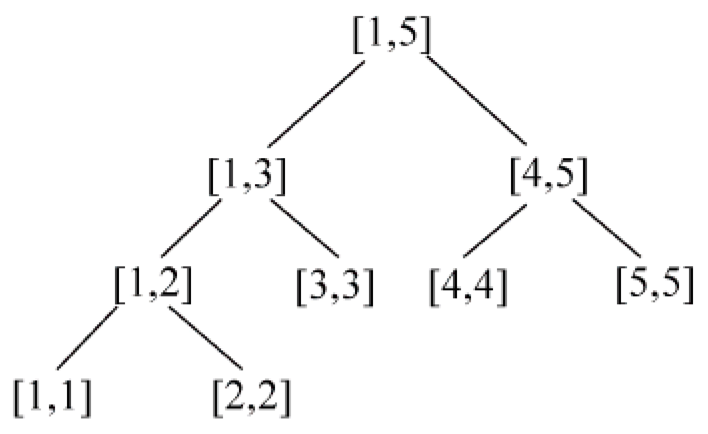


图 5.21 线段[1, 5]的线段树结构

考查每个线段$[L,R]$,L 是左子结点,R 是右子结点。

(1) $L=R$,说明这个结点只有一个点,它就是一个叶子结点。

(2) $L<R$,说明这个结点代表的不止一个点,它有两个儿子,左儿子代表的区间是$[L, M]$,右儿子代表的区间是$[M+1, R]$,其中 $M=(L+R)/2$。

线段树是二叉树,一个区间每次被折一半往下分,所以最多分 $\log_2 n$ 次就到达最低层。当需要查找一个点或者区间的时候,顺着结点往下找,最多 $\log_2 n$ 次就能找到。这就是线段树效率高的原因,使用了二叉树折半查找的方法。

回到 RMQ 问题,如果用线段树,“修改元素+查询最值”这两个操作分别可以在 $O(\log_2 n)$的时间内完成。如图 5.22 所示,查询{1,2,5,8,6,4,3}的最小值,其中每个结点上的数字是这棵子树的最小值。

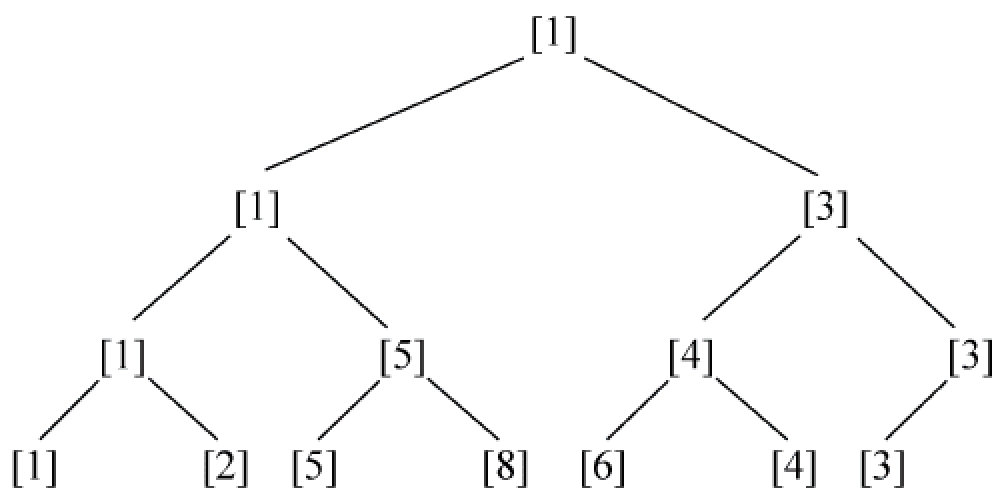


图 5.22　RMQ 问题(查询最小值)

如需修改元素,直接修改叶子结点上元素的值后从底往上更新线段树,操作次数也是 $O(\log_2 n)$。

m 次"修改+查询"的总复杂度是 $O(m\log_2 n\log_2 n)$。实际上,修改和查询可以同时做,所以总复杂度是 $O(m\log_2 n)$。这对规模 100 万的问题也能轻松解决。

5.3.2　点修改

首先讨论在线段树中每次只修改一个点的问题。

线段树如何构造?如何更新?如何查询?下面以 poj 2182 为例,引导出线段树的应用和编程细节。

> **poj 2182 "Lost Cows"**
>
> **题目描述**:有编号是 1~n 的 n 个数字,$2\leqslant n\leqslant 8000$,乱序排列,顺序是未知的。对于每个位置的数字,知道排在它前面比它小的数字有多少个。求这个乱序数列的顺序。

例如有 5 个数,已知每个数字前面比它小的数的个数,分别是:

pre[]:　0 1 2 1 0

可以求得这个乱序排列是:

ans[]:　2 4 5 3 1

本题是"简单题",用线段树或者树状数组实现。

在讲解后续内容之前,这里先用暴力法实现,思路是从后往前处理 pre[]:

(1) pre[5]=0,表示 ans[5]前面比它小的数有 0 个,即 ans[5]是最小的,在 1~5 这几个编号中 1 最小,所以 ans[5]=1。ans[]的前 4 个编号不再包括 1,剩下 2~5 这几个编号。

(2) pre[4]=1,在剩下的 2~5 这几个编号中,编号 3 是第 2 大的,所以 ans[4]=3。

(3) pre[3]=2,在剩下的 2、4、5 这几个编号中,编号 5 是第 3 大的,所以 ans[3]=5。

(4) pre[2]=1,在剩下的 2、4 这两个编号中,编号 4 是第 2 大的,所以 ans[2]=4。

(5) pre[1]=0,剩下的编号 2,ans[1]=2。

概括以上步骤,在每一步,剩下的编号中第 pre[n]+1 大的编号就是 ans[n]。

用暴力的方法,从 pre[]末尾往前计算,每处理一头牛后,需要把剩下的牛重新排名,重新排名的计算时间是 $O(n)$;在重新排名时,可以顺便做下一次的查找,所以不需要另外算查找的时间。一共有 n 头牛,总复杂度是 $O(n^2)$。本题的数据规模 n 不大,只有 8000,用暴力的方法也能通过。

在下面的代码中，pre[]是输入的 $1\sim n$ 个数字，例如{0 1 2 1 0}；ans[]是答案，例如{2 4 5 3 1}；num[]记录被处理过的数字，被处理后的数字置为−1。例如 num[]的初始值是{1 2 3 4 5}，得到 ans[5]=1 后，num[]更新为{−1 2 3 4 5}，这里用−1 表示 1 这个数字已经用过了。

解题的关键是，在剩下的编号中，第 pre[n]+1 个数字就是 ans[n]。

下面是代码。

poj 2182 的暴力法代码

```
#include <stdio.h>
const int Max = 8005;
int main(){
    int n, i, j, k;
    int pre[Max], ans[Max], num[Max];        //数组的第 0 个都不用,从第 1 个开始用
    scanf("%d", &n);
    pre[1] = 0;
    for(i=1; i<=n; i++)       num[i] = i;
    for(i = 2; i<=n; i ++)    scanf("%d", &pre[i]);
    for(i = n; i>=1; i-- ) {                 //从后往前处理数列
        k=0;
        for(j=1; j<=n; j++)                  //查找 num[]中未处理的第 pre[i]+1 大的数
            if(num[j] != -1) {
                k++;
                if(k == pre[i]+1) {          //找到了
                        ans[i] = num[j];     //num[]中剩下的第 pre[i]+1 个数就是 ans[i]
                    num[j] = -1;
                    break;
                }
            }
    }
    for(i = 1; i<=n; i++)   printf("%d\n", ans[i]);
    return 0;
}
```

当 n 更大时，$O(n^2)$会 TLE，必须用更优的算法。问题的关键是，如何高效地对剩下的牛重新排名。

这里引入高级数据结构"线段树"，可以在 $O(\log_2 n)$的时间内完成一次重新排名。下面说明其要点。

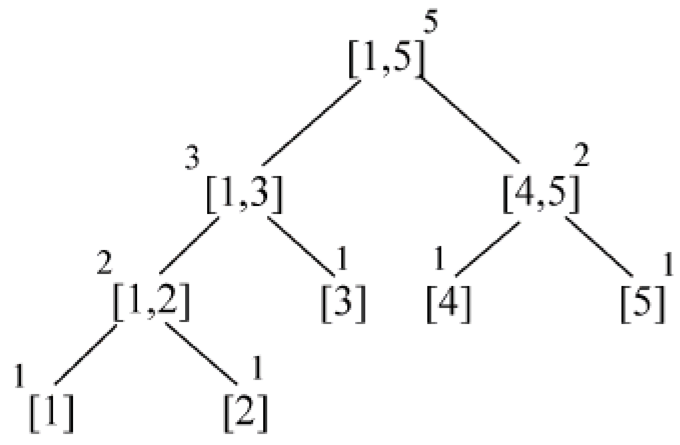


图 5.23　初始线段树

1. 用二叉树建立线段树

用二叉树的方法，把牛分成不同的组。在图 5.23 中，叶子结点内的数字是牛的编号，其他结点是牛的编号范围。例如根结点，包含 5 头牛，它的左子结点有 3 头牛，右子结点有两头牛。

2. 存储空间

如果牛有 n 头，这个二叉树的结点总数在编程时为

$4n$。请读者自己思考为什么是 $4n$(在后面的程序注释中有答案)。

3. 查询和更新

(1) 第 1 次处理 pre[5]=0,即找对应的第 1 头牛,如图 5.24(a)所示。步骤是从根结点开始,逐步找到左下角,即编号为 1 的结点,得到 ans[5]=1。在这个过程中,更新经过的每个结点,即把这个结点剩下的牛的数量减一。一共需要更新 4 个结点。左下角结点已经减到 0,表示后面的计算需要排除它。

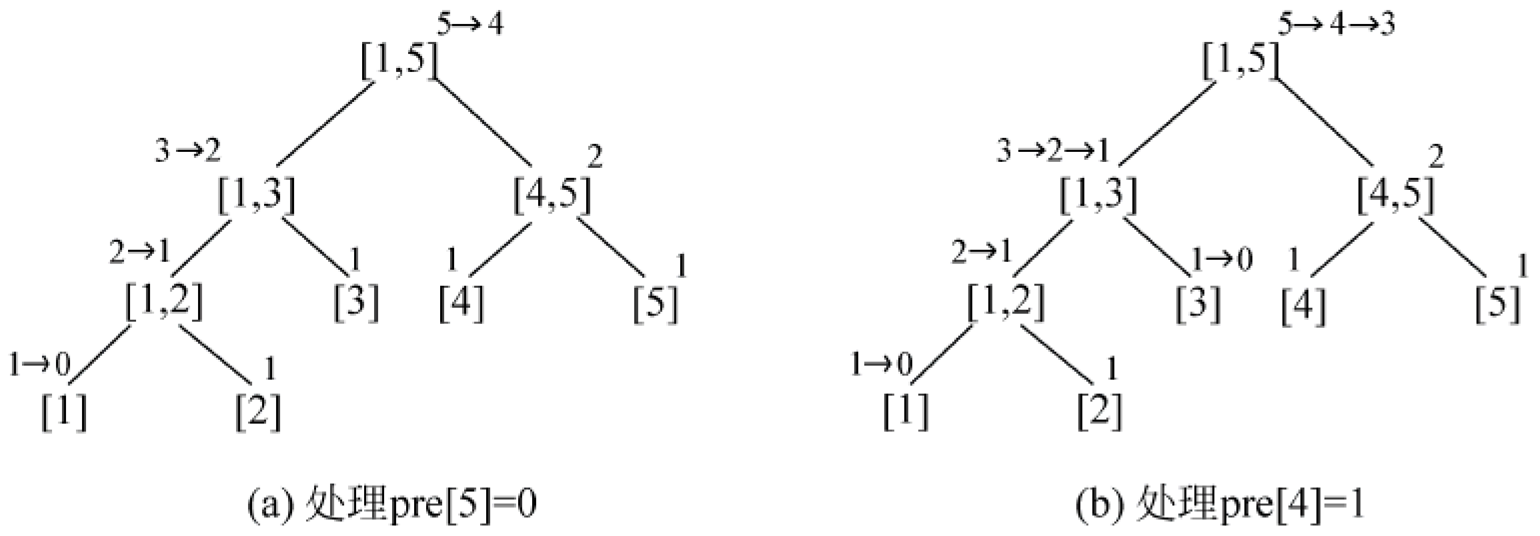


图 5.24　线段树的查询和更新

(2) 第 2 次处理 pre[4]=1,即找剩下的第 2 头牛,如图 5.24(b)所示。步骤是从根结点开始,逐步找到左边第 3 个结点,得到 ans[5]=3。更新经过的每个结点,一共更新 3 个结点。

(3) 依次处理,直到结束。

4. 复杂度

每次处理,从二叉树的根结点开始到最下一层,最多需要更新 $\log_2 4n$ 个结点,复杂度是 $O(\log_2 n)$;一共有 n 头牛需要处理,总复杂度是 $O(n\log_2 n)$。在暴力法中,每次需要查询和更新 n 个序列中的每个数,复杂度为 $O(n)$。线段树把 n 个数按二叉树进行分组,每次更新有关的结点时,这个结点下面的所有子结点都隐含被更新了,从而大大地减少了处理次数。

下面给出 poj 2182 的线段树代码。

poj 2182 “用结构体实现线段树”

```
#include<stdio.h>
using namespace std;
const int Max = 10000;
struct{
    int l, r, len;                    //用 len 存储这个区间的数字个数,即这个结点下牛的数量
}tree[4*Max];                          //这里是 4 倍,因为线段树的空间需要
int pre[Max], ans[Max];
void BuildTree(int left, int right, int u){ //建树
    tree[u].l = left;
    tree[u].r = right;
    tree[u].len = right - left + 1;        //更新结点 u 的值
     if(left == right)
         return;
    BuildTree(left, (left+right)>>1, u<<1);              //递归左子树
    BuildTree(((left+right)>>1)+1, right, (u<<1)+1);     //递归右子树
```

```
}
int query(int u, int num){                  //查询 + 维护,所求值为当前区间中左起第 num 个元素
    tree[u].len --;                         //对访问到的区间维护 len,即把这个结点上牛的数量减一
    if(tree[u].l == tree[u].r)
        return tree[u].l;
//情况 1:左子区间内牛的个数不够,则查询右子区间中左起第 num - tree[u<<1].len 个元素
    if(tree[u<<1].len < num)
        return query((u<<1) + 1, num - tree[u<<1].len);
//情况 2:左子区间内牛的个数足够,依旧查询左子区间中左起第 num 个元素
    if(tree[u<<1].len >= num)
        return query(u<<1, num);
}
int main(){
    int n, i;
    scanf("%d", &n);
    pre[1] = 0;
    for(i = 2; i <= n; i ++)
        scanf("%d", &pre[i]);
    BuildTree(1, n, 1);
    for(i = n; i >= 1; i --)                  //从后往前推断出每次最后一个数字
        ans[i] = query(1, pre[i] + 1);
    for(i = 1; i <= n; i ++)
        printf("%d\n", ans[i]);
    return 0;
}
```

5. 用完全二叉树实现线段树

在上面的例子中,线段树是一棵普通的二叉树,操作起来比较麻烦。其实可以用完全二叉树的结构来实现,编程更加简单,如图 5.25 所示。

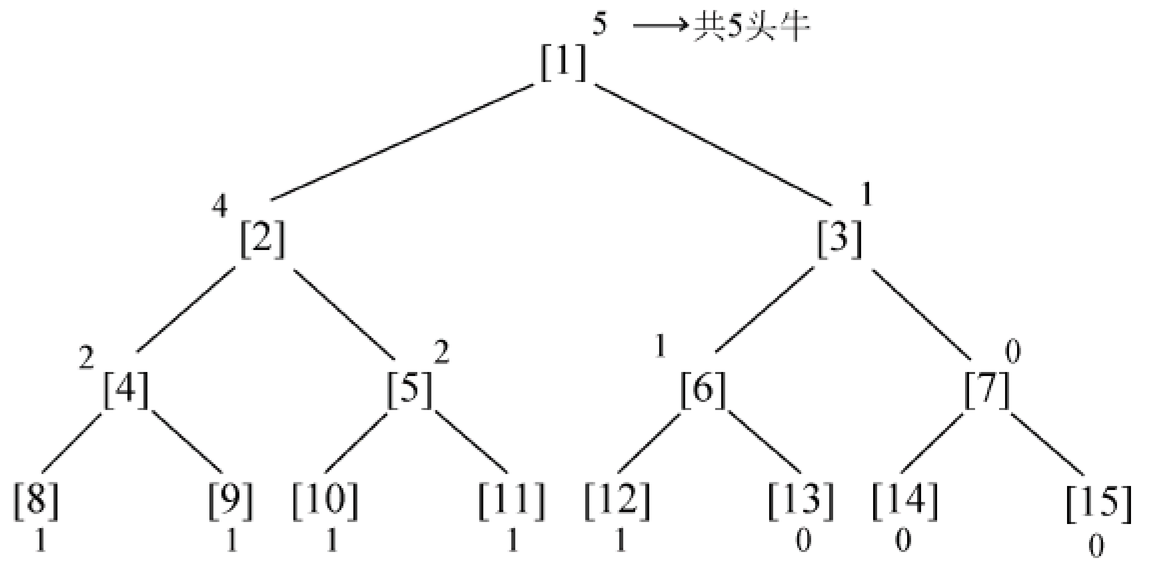


图 5.25 用完全二叉树建线段树

该图中的最后一行是牛的编号,例如[8]对应 1 号牛,[9]对应 2 号牛,等等。一共有 5 头牛。

在使用完全二叉树时,最后一层会存在"空叶子"。同样给空叶子按顺序编号,在遍历线段树时根据判断条件跳过这些"空叶子"就好了。用完全二叉树的方式存储线段树能提高插入线段和搜索时的效率。父结点 p 的左、右子结点分别是 $p*2$、$p*2+1$,用这样的索引方式检索 p 的左、右子树比用指针快。

poj 2182“用完全二叉树实现线段树”

```
#include <stdio.h>
#include <math.h>
const int Max = 10000;
int pre[Max] = {0}, tree[4 * Max] = {0},ans[Max] = {0};
//tree 是用数组实现的满二叉树.从图 5.25 可以知道,需要 4 倍大的空间
void BuildTree(int n,int last_left){        //用完全二叉树建一个线段树
    int i;
    for(i = last_left;i < last_left + n;i++)
                    //给二叉树的最后一行赋值,左边 n 个结点是 n 头牛
        tree[i] = 1;
    while(last_left != 1) {      //从二叉树的最后一行倒推到根结点,根结点的值是牛的总数
        for(i = last_left/2; i < last_left; i++)
            tree[i] = tree[i * 2] + tree[i * 2 + 1];
        last_left = last_left/2;
    }
}
int query(int u, int num,int last_left){
    //查询 + 维护,关键的一点是所求值为当前区间中左起第 num 个元素
    tree[u]--;                          //对访问到的区间维护剩下的牛的个数
    if(tree[u] == 0 && u >= last_left)
        return u;
//情况 1:左子区间的数字个数不够,则查询右子区间中左起第 num - tree[u<<1]个元素
    if(tree[u<<1] < num)
        return query((u<<1) + 1, num - tree[u<<1], last_left);
//情况 2:左子区间的数字个数足够,依旧查询左子区间中左起第 num 个元素
    if(tree[u<<1] >= num)
        return query(u<<1, num,last_left);
}
int main(){
    int n, last_left,i;
    scanf("%d", &n);
    pre[1] = 0;
    last_left = 1<<(int(log(n)/log(2)) + 1);
//二叉树最后一行的最左边一个.计算方法是找离 n 最近的 2 的指数,例如 3->4, 4->4, 5->8
    for(i = 2; i <= n; i ++)
        scanf("%d", &pre[i]);
    BuildTree(n,last_left);
    for(i = n; i >= 1; i--)              //从后往前推断出每次最后一个数字
        ans[i] = query(1, pre[i] + 1,last_left) - last_left + 1;
    for(i = 1; i <= n; i ++)
        printf("%d\n", ans[i]);
    return 0;
}
```

5.3.3　离散化

建二叉树是线段树的基本操作,但是二叉树的大小并不是无限制的,例如规模 10 000 000

以上的二叉树会超过允许的存储空间。在竞赛中如果出现结点规模这样大的题目，当然不能在程序中建这么大的二叉树，此时需要用“离散化”这种小技巧来解决。

离散化就是把原有的大二叉树压缩成小二叉树，但是压缩前后子区间的关系不变。

例如一块宣传栏，横向长度的刻度标记为 1 到 10，贴 4 张不同颜色的海报，它们的宽度和宣传栏等宽，长度分别是[1,3]、[2,5]、[3,8]、[3,10]，并且用后者覆盖前者，问最后能看见几种颜色的海报。

离散化步骤如下：

(1) 提取这 4 张海报的 8 个端点：1 3 2 5 3 8 3 10

(2) 排序并且删除相同的端点，得到：1 2 3 5 8 10

(3) 把原线段的 8 个端点映射到新的线段上：

1	2	3	5	8	10
↓	↓	↓	↓	↓	↓
1	2	3	4	5	6

新的 4 个海报为[1,3]、[2,4]、[3,5]、[3,6]，覆盖关系没有改变。新的宣传栏长度是 1 到 6，即宣传栏的长度从 10 压缩到 6。

离散化的压缩比是很可观的。例如原线段树的区间长度是 10 000 000，而其中真正用到的子区间是 100 000，那么子区间的端点最多有 2×100 000 个。经过离散化压缩后，新的线段树区间是 200 000，压缩率是 200 000/10 000 000=2%。

【习题】

poj 2528，题目中宣传栏的长度是 10 000 000。

5.3.4 区间修改

上面的例子都是只修改线段树上的某个点。区间修改是更复杂的问题。给定 n 个元素 $\{a_1,a_2,\cdots,a_n\}$，进行以下操作：

加：给定 $i,j\leqslant n$，把$\{a_i,\cdots,a_j\}$区间内的值全部加 v。

查询：给定 $L,R\leqslant n$，计算$\{a_L,\cdots,a_R\}$的区间和。

下面以 poj 3468 为例来讲解区间修改问题。

poj 3468 “A Simple Problem with Integers”

给出 N 个数，进行 Q 个操作，$1\leqslant N, Q\leqslant 100\,000$。有两种操作：

“C a b c”，对区间[a,b]的每个数字加 c；

“Q a b”，查询区间[a,b]的数字和。

输入：N，Q，以及 N 个数字，Q 个操作；

输出：对每个查询操作，输出结果。

如果用暴力方法，直接对这 n 个数进行操作，那么每个 C 操作和 Q 操作都是 $O(n)$的，一共有 Q 次操作，总复杂度是 $O(n^2)$。

如果用前面的修改线段树点的方法，在做 C 操作时，对区间里的数一个一个进行修改，

一个数的修改是 $O(\log_2 n)$ 的，区间修改合起来是 $O(n\log_2 n)$，Q 次操作的总复杂度是 $O(n^2\log_2 n)$，比暴力法还要差。

lazy-tag 方法。此时可以采用一种“懒惰(lazy)”的做法。当修改的是一个整块区间时，只对这个线段区间进行整体上的修改，其内部每个元素的内容先不做修改，只有当这部分线段的一致性被破坏时才把变化值传递给子区间。那么，每次区间修改的复杂度是 $O(\log_2 n)$，一共有 Q 次操作，总复杂度是 $O(n\log_2 n)$。做 lazy 操作的子区间，需要记录状态(tag)，在下面的代码中用 add[]实现。

视频讲解

下面描述具体步骤。

(1) 初始化时建树。以区间[1, 10]为例建树，图 5.26 所示为结果。在最后的叶子上是 1～10 这 10 个数字。图中最底层有很多叶子是空的。每个结点右上角的数字是以它为根结点的这棵子树的区间和。

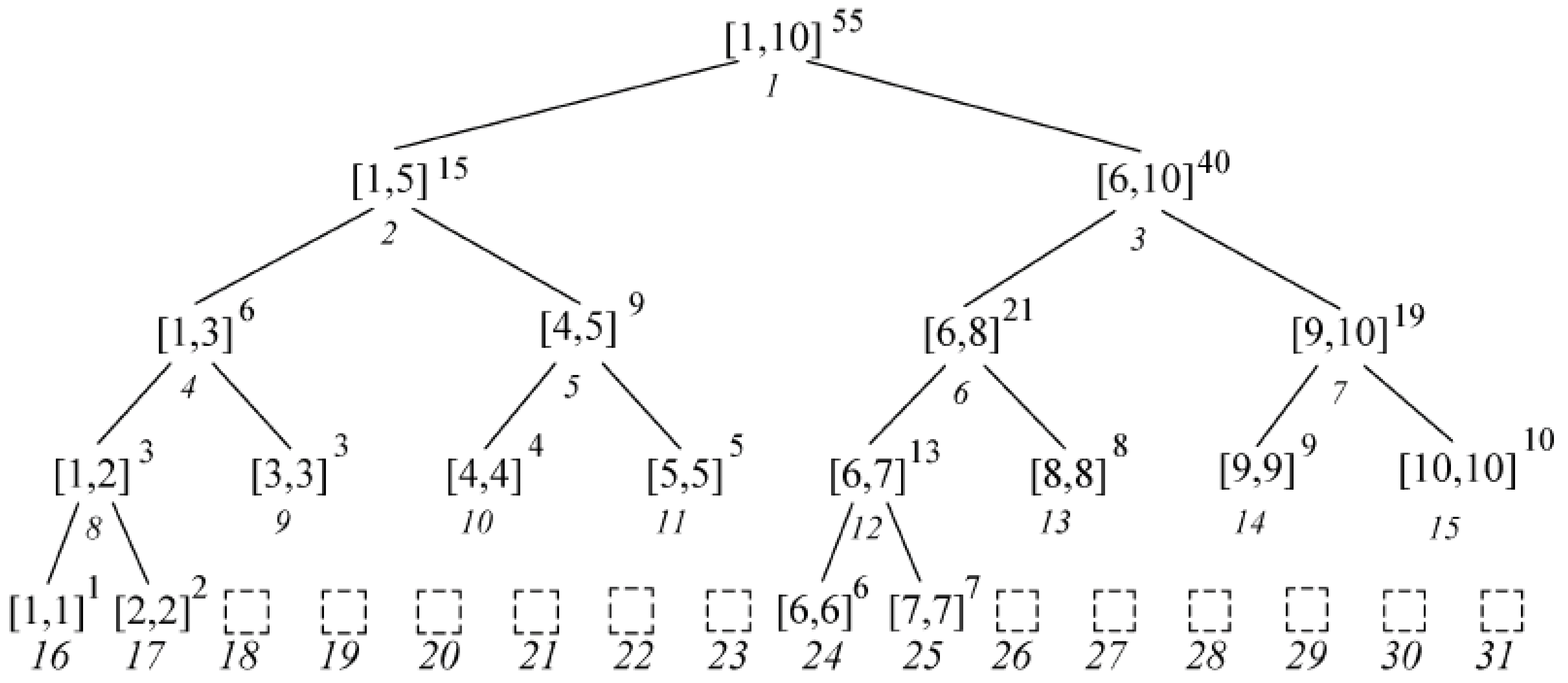


图 5.26　初始化建树

(2) “C *a b c*”操作。例如“C *3 6 3*”，在[3, 6]区间内，把每个元素加 3。从根结点开始，用递归在子树中找区间[3, 6]，有两种情况：[3, 6]与子区间交错、[3, 6]包含子区间。例如子区间[1, 5]和[6, 10]都与[3,6]交错，需要继续深入更底层子区间。在子区间[4, 5]，它被[3, 6]包含，那么根据 lazy 原理，把这个子区间进行整体修改，不继续深入，它下一层的[4, 4]和[5, 5]的区间和不用修改。图 5.27 所示为结果。部分结点的区间和发生了改变，见右上角。

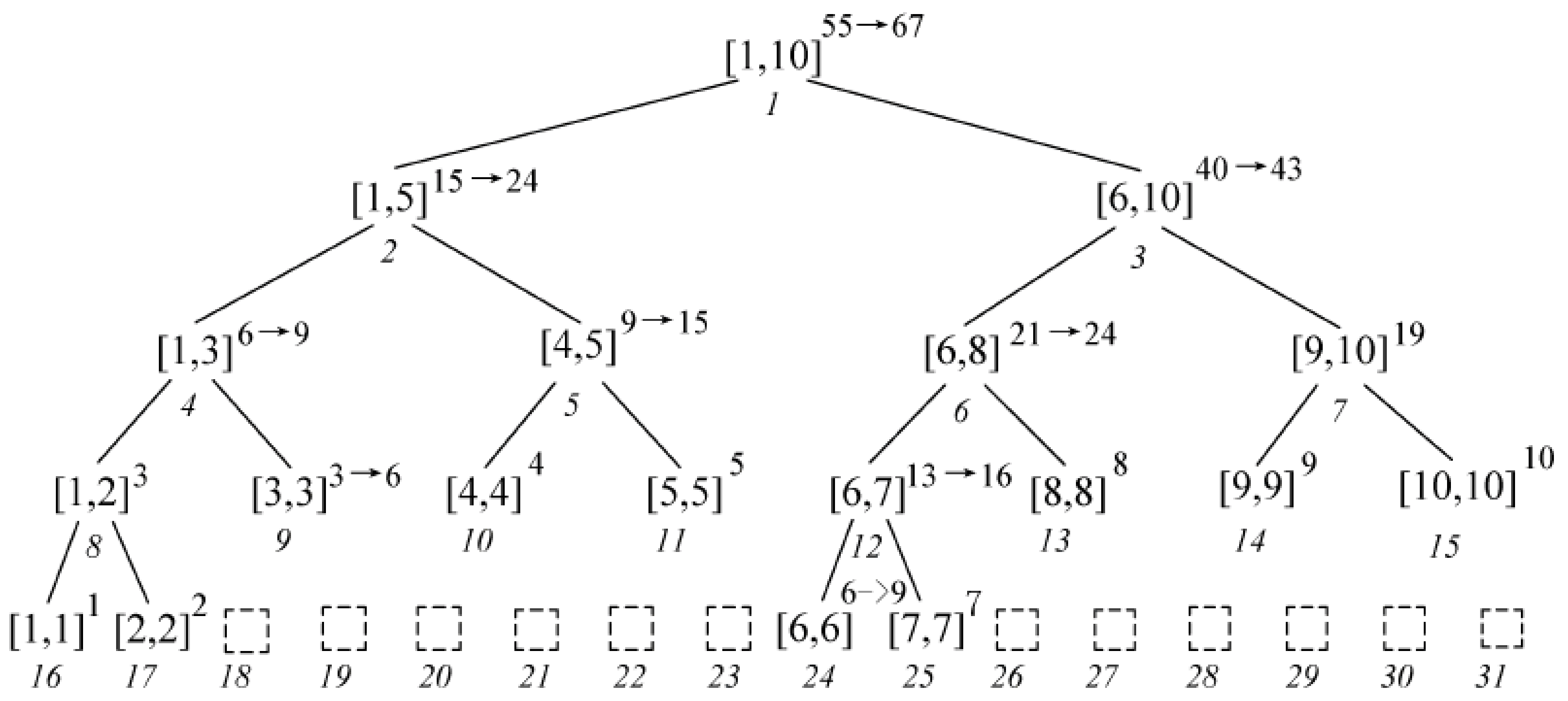


图 5.27　区间求和

(3)“Q a b”。同样可以利用 lazy 原理，当某个子区间包含在被查询的区间内时，直接返回这个子区间的区间和，不用继续深入。

下面是 poj 3468 的程序。build()函数建树，建树的结果见图 5.27；update()函数完成“C a b c”操作，query()函数完成“Q a b”操作。

sum[i]记录结点 i 的区间和，在图 5.27 中是结点右上角的数字。

add[i]是 tag，它记录结点 i 是否用到 lazy 原理，其值是“C a b c”中的 c；如果做了多次 lazy，add[i]可以累加。一旦结点 i 在某次“C a b c”中被深入，破坏了 lazy，就把 add[i]归零，push_down()函数完成这一任务。

```
#include <stdio.h>
using namespace std;
const int MAXN = 1e5 + 10;
long long sum[MAXN << 2], add[MAXN << 2];        //4 倍空间
void push_up(int rt){                            //向上更新,通过当前结点 rt 把值递归到父结点
    sum[rt] = sum[rt << 1] + sum[rt << 1 | 1];
}
void push_down(int rt, int m){                   //更新 rt 的子结点
    if(add[rt]){
        add[rt << 1] += add[rt];
        add[rt << 1 | 1] += add[rt];
        sum[rt << 1] += (m - (m >> 1)) * add[rt];
        sum[rt << 1 | 1] += (m >> 1) * add[rt];
        add[rt] = 0;                             //取消本层标记
    }
}
#define lson l, mid, rt << 1
#define rson mid + 1, r, rt << 1 | 1
void build(int l, int r, int rt){                //用满二叉树建树
    add[rt] = 0;
    if(l == r){                                  //叶子结点,赋值
        scanf("%lld", &sum[rt]);
        return;
    }
    int mid = (l + r) >> 1;
    build(lson);
    build(rson);
    push_up(rt);                                 //向上更新区间和
}
void update(int a, int b, long long c, int l, int r, int rt){      //区间更新
    if(a <= l && b >= r){
        sum[rt] += (r - l + 1) * c;
        add[rt] += c;
        return;
    }
    push_down(rt, r - l + 1);                    //向下更新
    int mid = (l + r) >> 1;
    if(a <= mid) update(a, b, c, lson);          //分成两半,继续深入
    if(b > mid) update(a, b, c, rson);
    push_up(rt);                                 //向上更新
}
```

```
long long query(int a, int b, int l, int r, int rt){    //区间求和
    if(a <= l && b >= r) return sum[rt];                  //满足 lazy,直接返回值
    push_down(rt, r - l + 1);                             //向下更新
    int mid = (l + r) >> 1;
    long long ans = 0;
    if(a <= mid) ans += query(a, b, lson);
    if(b > mid) ans += query(a, b, rson);
    return ans;
}
int main(void){
    int n, m;
    scanf("%d%d", &n, &m);
    build(1, n, 1);
    while(m--){
        char str[2];
        int a, b; long long c;
        scanf("%s", str);
        if(str[0] == 'C'){
            scanf("%d%d%lld", &a, &b, &c);
            update(a, b, c, 1, n, 1);
        }else{
            scanf("%d%d", &a, &b);
            printf("%lld\n", query(a, b, 1, n, 1));
        }
    }
}
```

5.3.5　线段树习题

简单题：hdu 1166/1394/1698/1754/2795；

poj 1195/2182/2299/2828/2352/2750/2886/2777/3264/3468。

中等题：hdu 1540/1823/4027/5869；

poj 2155/2528/2823/3225。

综合题：hdu 1255/1542/3642/3974/4578/4614/4718/5756/4441。

5.4　树状数组

树状数组(Binary Indexed Tree,BIT)是一种利用数的二进制特征进行检索的树状结构。树状数组是一种奇妙的数据结构,不仅非常高效,而且代码极其简洁。

1. 树状数组的概念

从下面这个例子引导出树状数组的概念。

长度为 n 的数列$\{a_1,a_2,\cdots,a_n\}$,进行以下操作。

(1) 修改元素 add(k,x)：把 a_k 加上 x。

(2) 求和 sum(x)：$x\leqslant n$,sum$=a_1+a_2+\cdots+a_x$。那么,区间和 $a_i+\cdots+a_j=\text{sum}(j)-\text{sum}(i-1)$。

这个程序很好写，用循环加或者前缀和，复杂度是 $O(n)$。然而，如果 n 很大，这样做的效率会非常低。读者可以用前面讲的线段树来实现高效的算法。其实有一种更好的数据结构，即树状数组，不仅效率和线段树一样高，只有 $O(\log_2 n)$，而且代码短得不可思议。先看一看代码：

```
#define lowbit(x)  ((x) & - (x))
void add(int x, int d) {                    //更新数组 tree[]。ax = ax + d，修改和 ax 有关的 tree[]
    while(x <= n) {
        tree[x]  += d;
        x += lowbit(x);
    }
}
int sum(int x) {                            //求和：sum = a1 + a2 + … + ax
    int sum = 0;
    while(x > 0){
        sum += tree[x];
        x -= lowbit(x);
    }
    return sum;
}
```

add()和 sum()的复杂度都是 $O(\log_2 n)$。

上述代码的使用方法如下：

(1) 初始化，add()。先清空数组 tree[]，然后读取 $a_1, a_2, \cdots, a_n$，用 add()逐一处理这 n 个数，得到 tree[]数组。在程序中并不需要定义数组 a[]，因为它隐含在 tree[]中。

(2) 求和，sum()。计算 $\text{sum}=a_1+a_2+\cdots+a_x$，即执行 sum()。求和是基于数组 tree[]的。

(3) 如果需要修改元素，执行 add()，即修改数组 tree[]。

下面详细说明上述操作的原理。

2. lowbit()操作

从代码中可以看出，其核心是一个神奇的 lowbit(x)操作。lowbit(x)＝x & $-x$，功能是找到 **x 的二进制数的最后一个 1**。其原理是利用负数的补码表示，补码是原码取反加一。例如 $x=6=00000110_2$，$-x=x_{补}=11111010_2$，那么 lowbit(x)$=x$ & $-x=10_2=2$。

视频讲解

1～9 的 lowbit()结果如表 5.1 所示。

表 5.1　1～9 的 lowbit()结果

x	1	2	3	4	5	6	7	8	9
x 的二进制	1	10	11	100	101	110	111	1000	1001
lowbit(x)	1	2	1	4	1	2	1	8	1
tree[x]数组	tree[1] $=a_1$	tree[2] $=a_1+a_2$	tree[3] $=a_3$	tree[4] $=a_1+a_2+a_3+a_4$	tree[5] $=a_5$	tree[6] $=a_5+a_6$	tree[7] $=a_7$	tree[8] $=a_1+a_2+\cdots+a_8$	tree[9] $=a_9$

lowbit(x)有什么用呢？从 lowbit(x)引出一个 tree[]数组，所有的计算都围绕 tree[]进行。

令 m=lowbit(x)，定义 tree[x]的值，是把 a_x 和它前面共 m 个数相加的结果，如表 5.1 所示。例如 lowbit(6)=2，tree[6]=a_5+a_6。

图 5.28 中的横线重新描述了这个关系，横线中的黑色表示 tree[x]，它等于横线上元素相加的和。

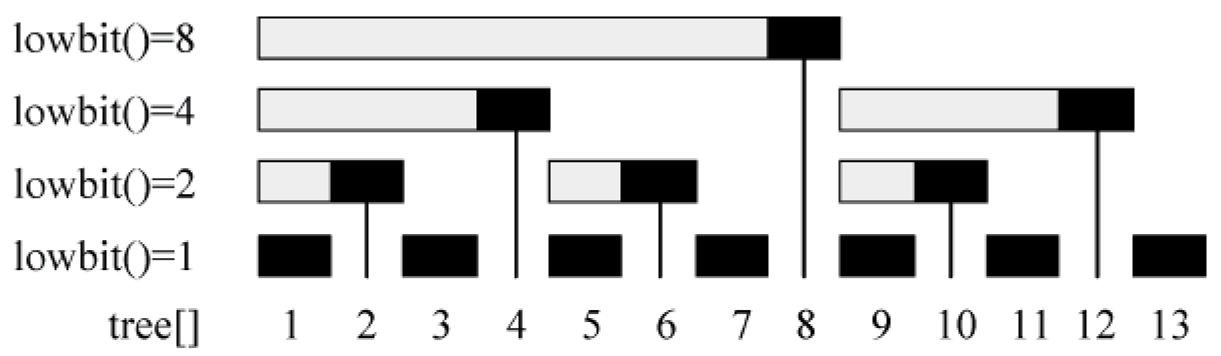


图 5.28 lowbit()计算

求和计算以及 tree[]数组的更新都可以通过 lowbit()完成。

1) 求和计算 sum=$a_1+a_2+\cdots+a_x$

可以借助 tree[]数组求 sum，例如：

sum(8)=tree[8]

sum[7]=tree[7]+tree[6]+tree[4]

sum[9]=tree[9]+tree[8]

然而，如何得到上面的关系呢？

很容易观察到，在计算 sum 时，对 tree[]的查找可以通过 lowbit(x)实现。例如 sum[7]=tree[7]+tree[6]+tree[4]。

首先从 7 开始，加上 tree[7]；

然后 7−lowbit(7)=6，加上 tree[6]；

接着 6−lowbit(6)=4，加上 tree[4]；

最后 4−lowbit(4)=0，结束。

编程细节见前面的求和函数 sum()，复杂度是 $O(\log_2 n)$。

2) tree[]数组的更新

更改 a_x，那么和它相关的 tree[]都会变化。例如改变 a_3，那么 tree[3]、tree[4]、tree[8]等都会改变。同样，这个计算也利用了 lowbit(x)。

首先更改 tree[3]；

然后 3+lowbit(3)=4，更改 tree[4]；

接着 4+lowbit(4)=8，更改 tree[8]；

继续，直到最后的 tree[n]。

编程细节见函数 add()，复杂度也是 $O(\log_2 n)$。add()函数也用于 tree[]的初始化过程：tree[]初始化为 0，然后用 add()逐一处理 $a_1,a_2,\cdots,a_n$。

3. 例题

这里仍然以 poj 2182 为例，用树状数组实现。该题用树状数组更容易理解。

其中的关键点如下：

(1) 在 n 个位置上，每个位置有一头牛，即 $a_1=a_2=\cdots=a_n=1$。不过，在程序中并不需要直接定义和使用数组 a[]。

(2) tree[]数组的初始化。这个题目比较特殊，不需要用 add()初始化，因为 lowbit(i)就是 tree[i]。

(3) 程序所做的，就是对每个 pre[i]+1，用 findpos()找出 sum(x)=pre[i]+1 所对应的 x，就是第 x 头牛。在找到第 x 头牛之后，令 $a_x=0$，方法是用 add()更新数组 tree[]，即执行 add(x, −1)。

下面的程序完全套用了上面提到的树状数组的模板。

poj 2182 "树状数组"

```
#include <stdio.h>
#include <string.h>
const int Max = 10000;
int tree[Max], pre[Max], ans[Max];
int n;
#define lowbit(x)  ((x) & - (x))
void add(int x, int d){
    while(x <= n) {
        tree[x] += d;
          x += lowbit(x);
    }
}
int sum(int x){
    int sum = 0;
    while(x > 0) {
        sum += tree[x];
          x -= lowbit(x);
    }
    return sum;
}
int findpos(int x){                    //寻找 sum(x) = pre[i] + 1 所对应的 x,就是第 x 头牛
    int l = 1, r = n;
    while(l < r) {
        int mid = (l + r) >> 1;
        if(sum(mid) < x)
            l = mid + 1;
        else
            r = mid;
    }
    return l;
}
int main(){
    scanf("%d",&n);
    pre[1] = 0;
    for(int i = 2; i <= n; i++)
        scanf("%d",&pre[i]);
    for(int i = 1; i <= n; i++)          //初始化 tree[]数组
          //注意这个题目比较特殊,不需要用 add()初始化,因为 lowbit(i)就是 tree[i]
```

```
        tree[i] = lowbit(i);
    for(int i = n; i > 0; i-- )  {
        int x = findpos(pre[i] + 1);
        add(x, - 1);                    //更新 tree[]数组
        ans[i] = x;
    }
    for(int i = 1; i <= n; i++)
        printf("%d\n", ans[i]);
    return 0;
}
```

4. 线段树和树状数组的对比

两者的复杂度同级，但是树状数组的常数明显优于线段树，编程复杂度也远远小于线段树。

线段树的适用范围大于树状数组，凡是可以使用树状数组解决的问题，使用线段树一定可以解决。树状数组的优点是编程非常简洁，使用 lowbit()可以在很短的几步操作中完成核心操作，代码效率远远高于线段树。

【习题】

简单题：poj 2299/2352/1195/2481/2029。

中等题：poj 2155/3321/1990；

　　　　hdu 3015/2430/2852。

难题：poj 2464，uva 11610。

5.5　小　　结

本章介绍了几个竞赛中常用的数据结构，限于篇幅，还有一些常用的数据结构没讲，例如堆、Hash、动态树 LCT 等。关于字符串的数据结构，在第 9 章中讲解；关于图的数据结构，在第 10 章中讲解。

高级数据结构是算法竞赛中比较难的内容，不仅本身的概念难以掌握，而且在具体的题目中需要根据情况灵活修改，以至于逻辑复杂、代码冗长。

第 6 章　基础算法思想

✍ 贪心法
✍ Huffman 编码
✍ 分治法
✍ 归并排序
✍ 快速排序
✍ 减治法

在竞赛中，队员拿到一个题目后很快就能知道这个题的考点是什么，例如图论、几何、数学、模拟、高级数据结构等。有时候老队员还会说："这一题的思路是动态规划……"

这里提到的动态规划并不是一个具体的算法，而是一种算法思想，或者是解题策略。类似地，把算法思想分成一些大类[①]，即暴力法、分治法、减治法、贪心法、动态规划。

本章将详细介绍贪心法、分治法、减治法。暴力法已经在"第 4 章 搜索技术"中介绍，动态规划将在"第 7 章 动态规划"中详细展开。

对于算法竞赛初学者来说，从只会按自然理解和逻辑做题，到能使用算法思想分析和设计，建立起基本的计算思维意识，是成为高级编程者的重要一步。

6.1　贪　心　法

6.1.1　基本概念

贪心(Greedy)是最容易理解的算法思想：把整个问题分解成多个步骤，在每个步骤都选取当前步骤的最优方案，直到所有步骤结束；在每一步都不考虑对后续步骤的影响，在后续步骤中也不再回头改变前面的选择。简单地说，其思想就是"走一步看一步""目光短浅"。

贪心法看起来似乎不靠谱，因为局部最优的组合不一定是全局最优的。那么，是否有一些规则使得局部最优能达到全局最优？本节将通过一些例子来详细说明这个问题。

贪心法有广泛的应用。例如图论中的最小生成树算法、单源最短路径算法 Dijkstra 是贪心思想的典型应用。关于这部分内容，请阅读"第 10 章 图论"。

下面先用硬币问题的例子引出贪心法的应用规则。

最少硬币问题：某人带着 3 种面值的硬币去购物，有 1 元、2 元、5 元的，硬币数量不限；他需要支付 M 元，问怎么支付才能使硬币数量最少？

---

① 讲解算法的经典教材《算法设计与分析基础》就是按这个分类展开的，由 Anany Levitin 著、潘彦译。另一本经典教材《算法导论》由 Thomas H. Cormen 等著、潘金贵等译，主要是按知识点内容来展开。

根据生活常识，第一步应该先拿出面值最大的 5 元硬币，第二步拿出面值第 2 大的 2 元硬币，最后才拿出面值最小的 1 元硬币。在这个解决方案中，硬币数量总数是最少的。

程序如下：

```
#include <bits/stdc++.h>
using namespace std;
const int NUM = 3;
const int Value[NUM] = {1,2,5};
int main(){
    int i, money;
    int ans[NUM] = {0};             //记录每种硬币的数量
    cin >> money;                   //输入钱数
    for(i = NUM-1; i>=0; i--){      //求每种硬币的数量
        ans[i] = money/Value[i];
        money = money - ans[i] * Value[i];
    }
    for(i = NUM-1; i>=0; i--)
        cout << Value[i] << "元硬币数:" << ans[i] << endl;
    return 0;
}
```

在上面的例子中，虽然每一步选硬币的操作并没有从整体最优来考虑，只在当前步骤选取了局部最优，但结果是全局最优的。然而，局部最优并不总是能导致全局最优。比如这个最少硬币问题，用贪心法一定能得到最优解吗？

在最少硬币问题中，如果稍微改一下参数，就不一定能得到最优解，甚至在有解的情况下也无法算出答案。

(1) 不能得到最优解的情形。例如，硬币面值比较奇怪，是 1、2、4、5、6 元，支付 9 元，如果用贪心法，答案是 6+2+1，需要 3 个硬币，而最优的 5+4 只需要两个硬币。

(2) 算不出答案的情形。例如，如果有面值 1 元的硬币，能保证用贪心法得到一个解，如果没有 1 元硬币，常常得不到解。用面值 2、3、5 元的硬币，支付 9 元，用贪心法无法得到解，但解是存在的，即 9=5+2+2。

所以，在最少硬币问题中是否能使用贪心法跟硬币的面值有关。如果是 1、2、5 这样的面值，贪心法是有效的，而对于 1、2、4、5、6 或者 2、3、5 这样的面值，贪心法是无效的①。对任意面值的硬币问题，需要用动态规划求最优解，在下一章讲解动态规划时会提到。

虽然贪心法不一定能得到最优解，但是它思路简单、编程容易。因此，如果一个问题确定用贪心法能得到最优解，那么应该使用它。

那么，如何判断一个题目能用贪心法？用贪心法求解的问题需要满足以下特征：

(1) 最优子结构性质。当一个问题的最优解包含其子问题的最优解时，称此问题具有最优子结构性质，也称此问题满足最优性原理。也就是说，从局部最优能扩展到全局最优。

① 一个简单的判断标准是，面值符合 $c_j > \sum_{i=1}^{i<j-1} c_i$ 的硬币，即任一面值的硬币，大于比它小的所有硬币的面值之和，可以用贪心法。例如以 2 的倍数递增的 1、2、4、8 等，这样的面值就符合条件。

(2) 贪心选择性质。问题的整体最优解可以通过一系列局部最优的选择来得到。

贪心算法没有固定的算法框架,关键是如何选择贪心策略。贪心策略必须具备无后效性,即某个状态以后的过程不会影响以前的状态,只与当前状态有关。

另外,对于某些难解问题,例如旅行商问题,很难得到最优解,但是此时用贪心法常常能得到不错的近似解。如果不一定非要求得最优解,那么贪心的结果也是很不错的方案。

6.1.2 常见问题

视频讲解

1. 活动安排问题

活动安排问题又称为区间调度问题,原型见 hdu 2037 题。

> **hdu 2037 "今年暑假不 AC"**
>
> 有很多电视节目,给出它们的起止时间,有的节目时间冲突,问能完整看完的电视节目最多有多少?

解题的关键在于选择什么贪心策略才能安排尽量多的活动。由于活动有开始时间和结束时间,考虑下面 3 种贪心策略:

(1) 最早开始时间。

(2) 最早结束时间。

(3) 用时最少。

经过分析发现,第 1 种策略是错误的,因为如果一个活动迟迟不终止,后面的活动就无法开始。第 2 种策略是合理的,一个尽快终止的活动可以容纳更多的后续活动。第 3 种策略也是错误的。

对最早结束时间进行贪心,算法步骤如下:

(1) 把 n 个活动按结束时间排序。

(2) 选择第 1 个结束的活动,并删除(或跳过)与它时间相冲突的活动。

(3) 重复步骤(2),直到活动为空。每次选择剩下的活动中最早结束的那个活动,并删除与它时间冲突的活动。

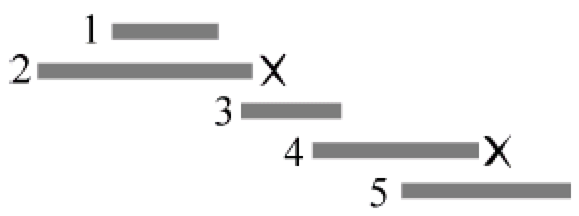


图 6.1 活动安排

下面的图 6.1 是例子,最优活动是 1、3、5,活动 2 和活动 4 与其他节目有冲突。

上述贪心算法是否能保证得到全局最优解?

(1) 它符合最优子结构性质。选中的第 1 个活动,它一定在某个最优解中;同理,选中的第 2 个活动、第 3 个活动等也都在这个最优解中。

(2) 它符合贪心选择性质。算法的每一步都使用了相同的贪心策略。

hdu 2037 部分代码

```
struct node {
    int start, end;                        //定义活动的起止时间
} record[MAXN];
bool cmp(const node& a, const node& b){return a.end < b.end; }
```

```
for(int i = 0; i < n; i++)                    //输入 n 个活动
    cin >> record[i].start >> record[i].end;
sort(record, record + n, cmp);                //按结束时间排序
int count = 0;
int lastend = -1;
for(int i = 0; i < n; i++) {                  //贪心算法
    if(record[i].start >= lastend){           //后一个起始时间大于等于前一个终止时间
        count++;
        lastend = record[i].end;              //记录前一个活动的终止时间
    }
}
cout << count << endl;                        //输出活动个数
```

2. 区间覆盖问题

给定一个长度为 n 的区间，再给出 m 条线段的左端点(起点)和右端点(终点)，问最少用多少条线段可以将整个区间完全覆盖？

贪心思路是尽量找出更长的线段。其解题步骤如下：

(1) 把每个线段按照左端点递增排序。

(2) 设已经覆盖的区间是$[L,R]$，在剩下的线段中找所有左端点小于等于 R 且右端点最大的线段，把这个线段加入到已覆盖区间里，并更新已覆盖区间的$[L,R]$值。

(3) 重复步骤(2)，直到区间全部覆盖。

在图 6.2 中，所有线段已按左端点进行排序。首先选中线段 1，然后在 2 和 3 中选中更长的 3。4 和 5 由于不合要求，被跳过。最后的最优解是 1、3。

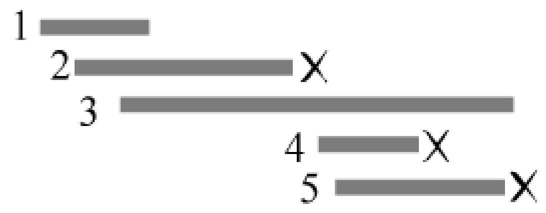


图 6.2　区间覆盖

3. 最优装载问题

原型见 hdu 2570 题。

> **hdu 2570 "迷瘴"**
>
> 有 n 种药水，体积都是 V，浓度不同，把它们混合起来，得到浓度不大于 $w\%$的药水，问怎么混合才能得到最大体积的药水？注意一种药水要么全用，要么都不用，不能只取一部分。

题目要求配置浓度不大于 $w\%$的药水，那么贪心的思路就是尽量找浓度小的药水。先对药水按浓度从小到大排序，药水的浓度不大于 $w\%$就加入，如果药水的浓度大于 $w\%$，计算混合后的总浓度，不大于 $w\%$就加入，否则结束判断。

4. 多机调度问题

设有 n 个独立的作业，由 m 台相同的计算机进行加工。作业 i 的处理时间为 t_i，每个作业可在任何一台计算机上加工处理，但不能间断、拆分。要求给出一种作业调度方案，在尽可能短的时间内，由 m 台计算机加工处理完成这 n 个作业。

求解多机调度问题的贪心策略是最长处理时间的作业优先，即把处理时间最长的作业分配给最先空闲的计算机。让处理时间长的作业得到优先处理，从而在整体上获得尽可能

短的处理时间。

(1) 如果 $n \leqslant m$，需要的时间就是 n 个作业当中最长的处理时间 t。

(2) 如果 $n > m$，首先将 n 个作业按处理时间从大到小排序，然后按顺序把作业分配给空闲的计算机。

6.1.3 Huffman 编码

Huffman 编码是贪心思想的典型应用，是一个很有用的、很著名的算法。Huffman 编码是“前缀”最优编码。

首先了解什么是编码。

把一段字符串存储在计算机中，这段字符串包含很多字符，每种字符出现的次数不一样，有的频次高，有的频次低。因为数据在计算机中都是用二进制码来表示的，所以需要把每个字符编码成一个二进制数。

最简单的编码方法是把每个字符都用相同长度的二进制数来表示。例如给出一段字符串，它只包含 A、B、C、D、E 这 5 种字符，编码方案如表 6.1 所示。

表 6.1　简单编码方案

字　符	A	B	C	D	E
频　次	3	9	6	15	19
编　码	000	001	010	011	100

每个字符用 3 位二进制数表示，存储的总长度是 $3\times(3+9+6+15+19)=156$。

这种编码方法简单、实用，但是不节省空间。由于每个字符出现的频次不同，可以想到用**变长编码**：出现次数多的字符用短码表示，出现少的用长码表示，例如表 6.2。

表 6.2　变长编码方案

字　符	A	B	C	D	E
频　次	3	9	6	15	19
编　码	1100	111	1101	10	0

存储的总长度是 $3\times4+9\times3+6\times4+15\times2+19\times1=112$。

第 2 种方法相当于第 1 种方法进行了压缩，压缩比是 $156/112=1.39$。

当然，编码算法的基本要求是编码后得到的二进制串能唯一地进行解码还原。上面第 1 种方法是正确的，每 3 位二进制数对应一个字符。第 2 种方法也是正确的，例如“11001111001101”，解码后唯一得到“ABDEC”。

如果胡乱设定编码方案，很可能是错误的，例如表 6.3。

表 6.3　错误编码方案

字　符	A	B	C	D	E
频　次	3	9	6	15	19
编　码	100	10	11	1	0

看起来似乎每个字符都有不同的编码，编码后的总长度也更短，只有 3×3＋9×2＋6×2＋15×1＋19×1＝73。但是编码无法解码还原，例如"100"，是"A"、"BE"还是"DEE"呢？

错误的原因是，某个编码是另一个编码的**前缀**(prefix)，即这两个编码有包含关系，导致了混淆。

那么有没有比第 2 种编码方法更好的方法？这引出了一个字符串存储的常见问题：给定一个字符串，如何编码，能使编码后的总长度最小？即如何得到一个最优解？

作为后续讲解的预习，读者可以验证：第 2 种编码方法已经达到了最优，编码后的总长度 112 就是能得到的最小长度。

下面介绍 Huffman 编码。Huffman 编码是前缀编码算法中的最优算法。

首先考虑如何进行编码？由于编码是二进制，容易想到用二叉树来构造编码。

例如上面第 2 种编码方案，其二叉树如图 6.3 所示。

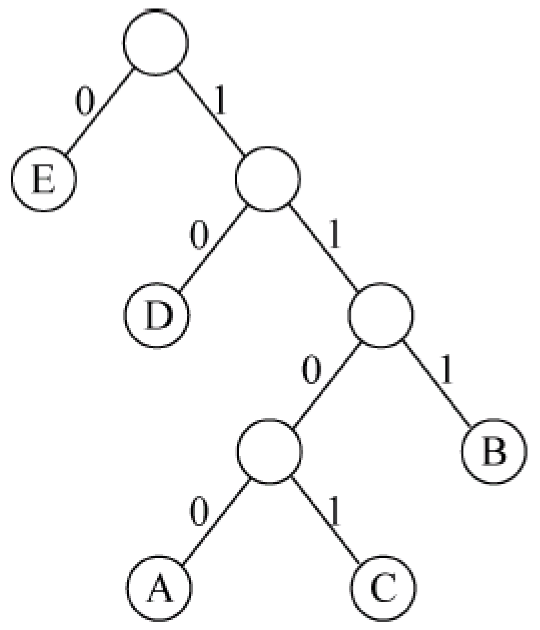


图 6.3　用二叉树实现前缀编码

在每个二叉树的分支，左边是 0，右边是 1。二叉树末端的叶子是编码，把编码放在叶子上，可以保证符合前缀不包含的要求。出现频次最高的字符 E，在最靠近根的位置，编码最短；出现频次最低的字符 A，在二叉树最深处，编码最长。

这棵编码二叉树是如何构造的？是最优的吗？

Huffman 编码是利用贪心思想构造二叉编码树的算法。

首先对所有字符按出现频次排序，如表 6.4 所示。

表 6.4　对字符按出现频次排序

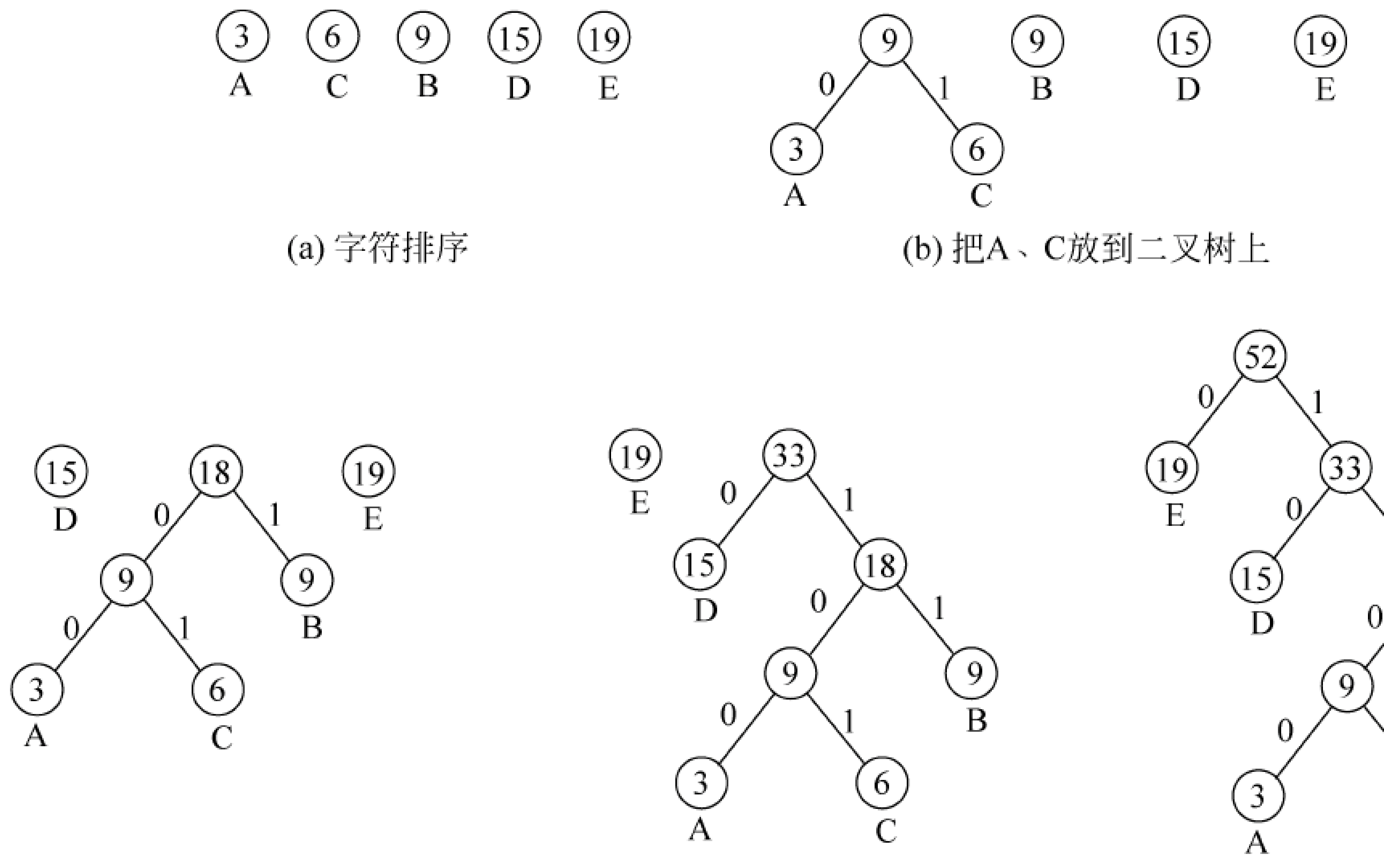
字　　符	A	C	B	D	E
频　　次	3	6	9	15	19

然后从出现频次最少的字符开始，用贪心思想安排在二叉树上。其步骤如图 6.4 所示。每个结点圆圈内的数字是这个子树下字符出现的频次之和。

贪心的过程是按出现频次从底层往顶层生成二叉树。注意，每一步都要按频次重新排序，例如图 6.4(c)和(d)中调整了 D 和 E 的顺序。这个过程可以保证出现频次少的字符被放在树的底层，编码更长；出现多的字符被放在上层，编码更短。

可以证明，Huffman 算法符合贪心法的“最优子结构性质”和“贪心选择性质”①。编码的结果是最优的。

① 证明见《算法导论》，Thomas H. Cormen 等著，潘金贵等译，机械工业出版社，234 页，“赫夫曼算法的正确性”。



(a) 字符排序　(b) 把A、C放到二叉树上　(c) 把B放到二叉树上，调整D　(d) 把D放到二叉树上，调整E　(e) 结果

视频讲解

图 6.4　Huffman 编码算法的步骤

下面给出一个例题。

> **poj 1521 "Entropy"**
>
> 输入一个字符串，分别用普通 ASCII 编码(每个字符 8bit)和 Huffman 编码，输出编码后的长度，并输出压缩比。
>
> 输入样例：
>
> AAAAABCD
>
> 输出样例：
>
> 64 13 4.9

这一题正常的解题过程是首先统计字符出现的频次，然后用 Huffman 算法编码，最后计算编码后的总长度。不过，由于只需要输出编码的总长度，而不要求输出每个字符的编码，所以可以跳过编码过程，利用图 6.4 描述的 Huffman 编码思想(圆圈内的数字是出现频次)，直接计算编码的总长度。

下面的代码使用了 STL 的优先队列，在每个贪心步骤，从优先队列中提取频次最低的两个字符。

poj 1521 部分代码

```
string s;
priority_queue<int, vector<int>, greater<int>> Q;
                                         //优先队列,最小的在队首
while(getline(cin, s) && s != "END"){    //输入字符串
    int t = 1;
    sort(s.begin(), s.end());
    for(int i=1;i<s.length();i++){       //统计字符出现的频次,并放进优先队列
```

```
            if(s[i] != s[i-1]){
                Q.push(t);
                t = 1;
            }
            else t++;
        }
        Q.push(t);
        int ans = 0;
        while(Q.size() > 1){
            int a = Q.top(); Q.pop();   //提取队列中最小的两个
            int b = Q.top(); Q.pop();
            Q.push(a+b);
            ans += a+b;                 //直接计算编码的总长度,请思考为什么
        }
        Q.pop();
    }
    //ans 就是编码后的总长度
```

6.1.4　模拟退火

模拟退火算法基于这样一个物理原理：一个高温物体降温到常温，温度越高时降温的概率越大(降温更快)，温度越低时降温的概率越小(降温更慢)。模拟退火算法利用这样一种思想进行搜索，即进行多次降温(迭代)，直到获得一个可行解。

在迭代过程中，模拟退火算法随机选择下一个状态，有两种可能：①新状态比原状态更优，那么接受这个新状态；②新状态更差，那么以一定的概率接受该状态，不过这个概率应该随着时间的推移逐渐降低。

模拟退火算法是贪心思想和概率的结合，常用“爬山”问题来介绍贪心有关的算法，在图 6.5 中，A 是局部最高点，B 是全局最高点。普通的贪心算法，如果当前状态在 A 附近，会一直爬山，最后停滞在局部最高点 A，而无法到达 B。模拟退火算法能跳出 A，得到 B。因为它不仅往上爬山，而且以一定的概率接受比当前点更低的点，使程序有机会摆脱局部最优到达全局最优。这个概率会随时间不断减小，从而最后能限制在最优解附近。

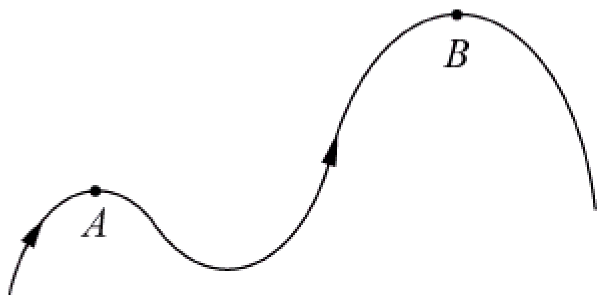


图 6.5　模拟退火与贪心

模拟退火算法的主要步骤如下：

(1) 设置一个初始的温度 T。

(2) 温度下降，状态转移。从当前温度按降温系数下降到下一个温度，在新的温度计算当前状态。

(3) 如果温度降到设定的温度下界，程序停止。

伪代码如下：

```
eps = 1e-8;                  //终止温度,接近 0,用于控制精度
T = 100;                     //初始温度,应该是高温,以 100℃ 为例
delta = 0.98;                //降温系数,控制退火的快慢,小于 1,以 0.98 为例
g(x);                        //状态 x 时的评价函数,例如物理意义上的能量
```

```
    now, next;                              //当前状态和新状态
    while(T > eps){                         //如果温度未降到 eps
            g(next), g(now);                //计算能量
            dE = g(next) - g(now);          //能量差
            if(dE >= 0)                     //新状态更优,接受新状态
                now = next;
            else if(exp(dE/T)> rand())      //如果新状态更差,在一定概率下接受它,e^(dE/T)
                now = next;
            T *= delta;                     //降温,模拟退火过程
    }
```

模拟退火在算法竞赛中的典型应用有函数最值问题、TSP 旅行商问题、最小圆覆盖、最小球覆盖等。在本书第 11.2.2 节中给出了用模拟退火求解最小圆覆盖的例子。下面的例子是求函数最值。

hdu 2899 "Strange function"

函数 $F(x)=6x^7+8x^6+7x^3+5x^2-yx$，其中 x 的范围是 $0\leqslant x\leqslant 100$。

输入 y 值，输出 $F(x)$ 的最小值。

用模拟退火求函数最值是最合适的。下面是代码：

```
#include<bits/stdc++.h>
using namespace std;
const double eps = 1e-8;                                //终止温度
double y;
double func(double x){                                  //计算函数值
    return 6 * pow(x,7.0) + 8 * pow(x,6.0) + 7 * pow(x,3.0) + 5 * pow(x,2.0) - y * x;
}
double solve(){
    double T = 100;                                     //初始温度
    double delta = 0.98;                                //降温系数
    double x = 50.0;                                    //x 的初始值
    double now = func(x);                               //计算初始函数值
    double ans = now;                                   //返回值
    while(T > eps){                                     //eps 是终止温度
        int f[2] = {1, -1};
        double newx = x + f[rand() % 2] * T;            //按概率改变 x,随 T 的降温而减少
        if(newx >= 0 && newx <= 100){
            double next = func(newx);
            ans = min(ans,next);
            if(now - next > eps){x = newx; now = next;} //更新 x
        }
        T *= delta;
    }
    return ans;
}
int main(){
    int cas; scanf("%d",&cas);
```

```
    while(cas -- ){
        scanf("%lf",&y);
        printf("%.4f\n",solve());
    }
}
```

模拟退火算法用起来非常简单、方便，不过也有缺点。它得到的是一个可行解，而不是精确解。例如上面的例题，计算到4位小数点的精度就停止，实际上是一个可行解，所以算法的效率和要求的精度有关。在一般情况下，模拟退火算法的复杂度会比其他精确算法差。用户在应用时需要仔细选择初始温度 T、降温系数 delta、终止温度 eps 等。

6.1.5　习题

hdu 1789 "Doing Homework again"，活动安排问题。

hdu 1050 "Moving Tables"，空间问题，模型和活动安排问题一样。

hdu 2546 "饭卡"，普通背包问题。

hdu 3348 "coins"，钱币问题。

hdu 4864 "task"，不错的题。

poj 1328 "Radar Installation"，几何问题，建模为活动安排问题。

poj 1089 "Intervals"，区间覆盖问题，给定很多线段，合并线段，使得合并后间隔最小。

6.2　分　治　法

分治法是广为人知的算法思想，很容易理解。人们在遇到一个难以直接解决的大问题时，自然会想到把它划分成一些规模较小的子问题，各个击破，"分而治之（Divide and Conquer）"。

在软件开发项目的详细设计阶段，常常会开一个"头脑风暴"会议，把整个项目分解成相对独立的子问题，其思想符合分治法。

分治算法的具体操作是把原问题分成 k 个较小规模的子问题，对这 k 个子问题分别求解。如果子问题不够小，那么把每个子问题再划分为规模更小的子问题。这样一直分解下去，直到问题足够小，很容易求出这些小问题的解为止。

能用分治法的题目需要符合以下两个特征。

（1）平衡子问题：子问题的规模大致相同，能把问题划分成大小差不多相等的 k 个子问题，最好 $k=2$，即分成两个规模相等的子问题。子问题规模相等的处理效率比子问题规模不等的处理效率要高。

（2）独立子问题：子问题之间相互独立。这是区别于动态规划算法的根本特征，在动态规划算法中，子问题是相互联系的，而不是相互独立的。

特别需要说明的是，分治法不仅能够让问题变得更容易理解和解决，而且能大大优化算法的复杂度，在一般情况下能把 $O(n)$ 的复杂度优化到 $O(\log_2 n)$。这是因为，**局部的优化有利于全局；一个子问题的解决，其影响力扩大了 k 倍，即扩大到了全局**。

举一个简单的例子：在一个有序的数列中查找一个数。简单的办法是从头找到尾，复杂度是 $O(n)$。如果用分治法，即"折半查找"，则最多只需要 $\log_2 n$ 次就能找到。

分治法是一种"并行"算法。由于子问题是相互独立的，因此可以把子问题分给不同的计算机，分开单独解决。

分治法如何编程？分治法的思想几乎就是递归的过程，用递归程序实现分治法是很自然的。

在用分治法建立模型时，解题步骤分为以下 3 步。

(1) 分解(Divide)：把问题分解成独立的子问题。

(2) 解决(Conquer)：递归解决子问题。

(3) 合并(Combine)：把子问题的结果合并成原问题的解。

分治法的经典应用有汉诺塔、快速排序、归并排序等。

6.2.1 归并排序

归并排序和快速排序都是非常精美的算法，学习它们，对于理解分治法思想、提高算法思维能力十分有帮助。在学习归并排序和快速排序之前，请读者先学习交换排序、选择排序、冒泡排序等暴力的排序方法①。

在介绍归并排序和快速排序之前先思考一个问题：如何用分治思想设计排序算法？

根据分治法的分解、解决、合并三步骤，具体思路如下：

(1) 分解。把原来无序的数列分成两部分，对每个部分，再继续分解成更小的两部分……在归并排序中，只是简单地把数列分成两半。在快速排序中，是把序列分成左、右两部分，左部分的元素都小于右部分的元素。分解操作是快速排序的核心操作。

(2) 解决。分解到最后不能再分解，排序。

(3) 合并。把每次分开的两个部分合并到一起。归并排序的核心操作是合并，其过程类似于交换排序。快速排序并不需要合并操作，因为在分解过程中左、右部分已经是有序的。

本节先讲解归并排序，然后讲解归并排序的典型应用——"逆序对"问题。

1. 归并排序示例

下面的例子给出了归并排序的操作步骤。初始数列经过 3 趟归并之后得到一个从小到大的有序数列，如图 6.6 所示。请读者根据这个例子分析它是如何实现分治法的分解、解决、合并 3 个步骤的。

分析该图，归并排序的主要操作如下：

(1) 分解。把初始序列分成长度相同的左、右两个子序列，然后把每个子序列再分成更小的两个子序列，直到子序列只包含 1 个数。这个过程用递归实现，图 6.6 中的第 1 行是初始序列，每个数是一个子序列，可以看成递归到达的最底层。

(2) 求解子问题，对子序列排序。最底层的子序列只包含 1 个数，其实不用排序。

---

① 算法竞赛中的排序，最多只处理千万级的数据量，即可以一次在内存中处理。工程上可能需要对大数据排序，例如 1TB 的数据，数据量太大，单个的 CPU 一次只能处理一小部分，所以不能简单地用某个排序算法。在找工作面试时，常常出现这种大数据排序的题目，读者可以学习有关的知识。

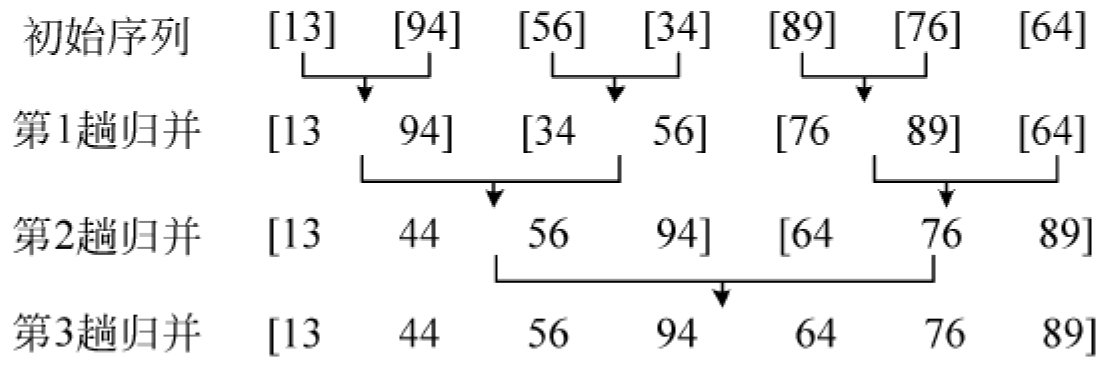
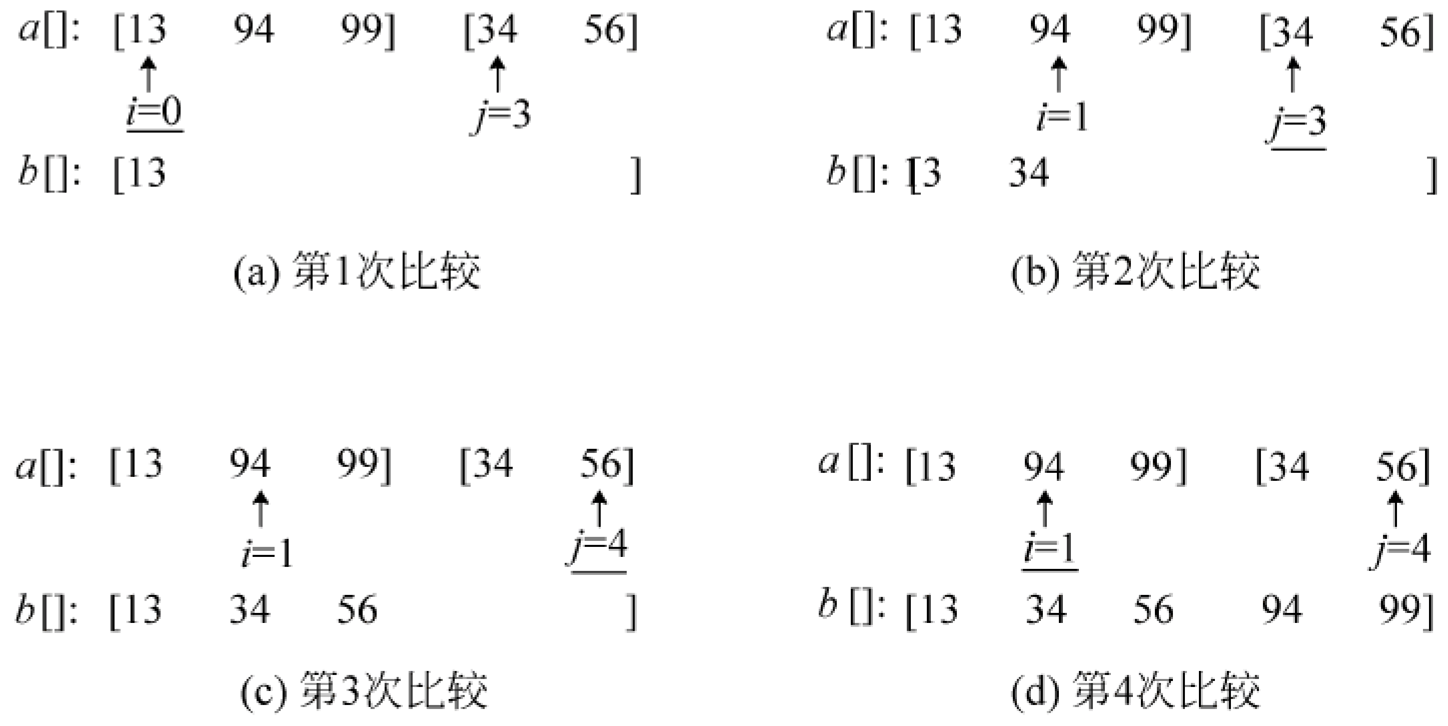


图 6.6　归并排序

(3) 合并。归并两个有序的子序列，这是归并排序的主要操作，过程如图 6.7 所示。例如在图 6.7(a)中，i 和 j 分别指向子序列{13,94,99}和{34,56}的第 1 个数，进行第 1 次比较，发现 $a[i]<a[j]$，把 $a[i]$放到临时空间 $b[]$中。总共经过 4 次比较，得到了 $b[]=\{13, 34,56,94,99\}$。



(a) 第1次比较　(b) 第2次比较　(c) 第3次比较　(d) 第4次比较

图 6.7　归并排序的一次合并

在暴力排序算法中，有一种算法是交换排序，归并排序可以看成是交换排序的升级版。

交换排序的步骤如下：

(1) 第 1 轮，检查第 1 个数 a_1。把序列中后面所有的数一个一个跟它比较，如果发现有一个比 a_1 小，就交换。第 1 轮结束后，最小的数就排在了第 1 个位置。

(2) 第 2 轮，检查第 2 个数。第 2 轮结束后，第 2 小的数排在了第 2 个位置。

(3) 继续上述过程，直到检查完最后一个数。

交换排序的复杂度是 $O(n^2)$。

在归并排序中，一次合并的操作和交换排序很相似，只是合并的操作是基于两个有序的子序列，效率更高。

下面分析归并排序的**计算复杂度**。对 n 个数进行归并排序：①需要 $\log_2 n$ 趟归并；②在每一趟归并中有很多次合并操作，一共需要 $O(n)$ 次比较。所以计算复杂度是 $O(n\log_2 n)$。

空间复杂度：由于需要一个临时的 $b[]$存储结果，所以空间复杂度是 $O(n)$。

读者从归并排序的例子可以体会到，对于整体上 $O(n)$复杂度的问题，通过分治可以减少为 $O(\log_2 n)$复杂度的问题。

2. 逆序对问题

排序是竞赛中的常用功能，一般直接使用 STL 的 sort()函数，并不需要自己再写一个

排序的程序。不过也有一些特殊的问题，需要写出程序，并在程序内部做一些处理，例如逆序对问题。

> **hdu 4911 "Inversion"**
>
> 输入一个序列$\{a_1, a_2, \cdots, a_n\}$，交换任意两个相邻元素，不超过 k 次。在交换之后，问最少的逆序对有多少个？
>
> 序列中的一个**逆序对**是指存在两个数 a_i 和 a_j，有 $a_i > a_j$ 且 $1 \leqslant i < j \leqslant n$。也就是说，大的数排在小的数前面。
>
> 输入：第 1 行是 n 和 k，$1 \leqslant n \leqslant 10^5$，$0 \leqslant k \leqslant 10^9$；第 2 行包括 n 个整数$\{a_1, a_2, a_3, \cdots, a_n\}$，$0 \leqslant a_i \leqslant 10^9$。
>
> 输出：最少的逆序对数量。
>
> 输入样例：
>
> 3 1
>
> 2 2 1
>
> 输出样例：
>
> 1

当 $k=0$ 时，就是求原始序列中有多少个逆序对。

求 $k=0$ 时的逆序对，用暴力法很容易：先检查第 1 个数 a_1，把后面的所有数跟它比较，如果发现有一个比 a_1 小，就是一个逆序对；再检查第 2 个数，第 3 个数……直到最后一个数。其复杂度是 $O(n^2)$。本题中 n 最大是 10^5，所以暴力法会 TLE。

考察暴力法的过程，会发现和交换排序很像。那么自然可以想到，能否用交换排序的升级版——归并排序来处理逆序对问题？

观察图 6.7 所示的一次合并过程发现，可以利用这个过程记录逆序对。观察到以下现象：

(1) 在子序列内部，元素都是有序的，不存在逆序对；逆序对只存在于不同的子序列之间。

(2) 在合并两个子序列时，如果前一个子序列的元素比后面子序列的元素小，那么不产生逆序对，如图 6.7(a)所示；如果前一个子序列的元素比后面子序列的元素大，就会产生逆序对，如图 6.7(b)所示。不过，在一次合并中，产生的逆序对不止一个，例如在图 6.7(b)中把 34 放到 b[]中时，它与 94、99 产生了两个逆序对。在下面的程序中，相关代码是"cnt+=mid-i+1;"。

根据以上观察，只要在归并排序过程中记录逆序对就行了。

以上解决了 $k=0$ 时原始序列中有多少个逆序对的问题，现在考虑，当 $k \neq 0$ 时(即把序列中任意两个相邻数交换**不超过** k 次)逆序对最少有多少？注意，不超过 k 次的意思是可以少于 k 次，而不是一定要 k 次。

在所有相邻数中，只有交换那些逆序的才会影响逆序对的数量。设原始序列有 cnt 个逆序对，讨论以下两种情况：

(1) 如果 cnt$\leqslant k$，总逆序数量不够交换 k 次。所以进行 k 次交换之后，最少的逆序对数

量为 0。

(2) 如果 cnt＞k，让 k 次交换都发生在逆序的相邻数上，那么剩余的逆序对是 cnt－k。

求逆序对的程序几乎可以完全套用归并排序的模板，差不多就是归并排序的裸题。在下面的程序中，Mergesort()和 Merge()是归并排序。与纯归并排序的程序相比，它只多了一句“cnt＋＝mid－i＋1;”。

hdu 4911 归并排序(求逆序对)

```
#include<bits/stdc++.h>
const int MAXN = 100005;
typedef  long  long  ll;
ll a[MAXN], b[MAXN], cnt;
void Merge(ll l, ll mid, ll r){
    ll i=l, j = mid+1, t=0;
    while(i<=mid && j<=r){
        if(a[i]>a[j]){
            b[t++] = a[j++];
            cnt += mid-i+1;                     //记录逆序对数量
        }
        else b[t++]=a[i++];
    }
  //一个子序列中的数都处理完了,另一个还没有,把剩下的直接复制过来
    while(i<=mid)    b[t++]=a[i++];
    while(j<=r)      b[t++]=a[j++];
    for(i=0; i<t; i++)  a[l+i] = b[i];          //把排好序的 b[]复制回 a[]
}
void Mergesort(ll l, ll r){
    if(l<r){
        ll  mid = (l+r)/2;                      //平分成两个子序列
        Mergesort(l, mid);
        Mergesort(mid+1, r);
        Merge(l, mid, r);                       //合并
    }
}
int main(){
    ll n,k;
    while(scanf("%lld%lld", &n, &k) != EOF){
        cnt = 0;
        for(ll i=0;i<n;i++)  scanf("%lld", &a[i]);
        Mergesort(0,n-1);                       //归并排序
        if(cnt<=k) printf("0\n");
        else             printf("%I64d\n", cnt - k);
    }
    return 0;
}
```

逆序对问题，除了可以用归并排序求解以外，也可以用树状数组求解。

6.2.2　快速排序

快速排序的思路是：把序列分成左、右两部分，使得左边所有的数都比右边的数小；递

归这个过程，直到不能再分为止。

那么如何把序列分成左、右两部分？最简单的办法是设定两个临时空间 X、Y 和一个基准数 t；检查序列中所有的元素，比 t 小的放在 X 中，比 t 大的放在 Y 中。其实不用这么麻烦，直接在原序列上操作就行了，不需要使用临时空间 X、Y。

直接在原序列上进行划分的方法也有很多种，下面的例子介绍了一种很容易操作的方法：

ij				t
5	2	8	3	4

尾部的 t 是基准数，i 指向比 t 小的左部分，j 指向比 t 大的右部分。

i	j			t
5	2	8	3	4

若 data[j]$\geqslant$data[t]，j++。

	i	j		t
2	5	8	3	4

若 data[j]<data[t]，交换 data[j]和 data[i]，然后 i++，j++。

		i	j	t
2	3	8	5	4

继续。

		i	j	t
2	3	4	5	8

最后，交换 data[i]和 data[t]，得到结果。i 指向基准数的当前位置。

下面用上述方法实现快速排序。

快速排序程序（poj 2388）

```
#include "stdio.h"
const int N = 10010;
int data[N];
#define swap(a, b) {int temp = a; a = b; b = temp;}      //交换
int partition(int left, int right){                    //划分成左、右两部分,以 i 指向的数为界
    int i = left;
    int temp = data[right];                            //把尾部的数看成基准数
    for(int j = left; j < right; j++)
        if(data[j] < temp){
            swap(data[j], data[i]);
            i++;
        }
    swap(data[i], data[right]);
     return i;                                         //返回基准数的位置
}
void quicksort(int left, int right){
    if(left < right){
        int i = partition(left, right);                //划分
          quicksort(left, i - 1);                      //分治:i 左边的继续递归划分
          quicksort(i + 1, right);                     //分治:i 右边的继续递归划分
    }
}
int main(){
```

```
    int n;
    scanf("%d", &n);
    for(int i = 1; i <= n; i++)   scanf("%d", &data[i]);
    quicksort(1, n);
    printf("%d\n", data[(n + 1)/2]);
    return 0;
}
```

下面分析**复杂度**。

每一次划分，都把序列分成了左、右两部分，在这个过程中，需要比较所有的元素，有 $O(n)$ 次。如果每次划分是对称的，也就是说左、右两部分的长度差不多，那么一共需要划分 $O(\log_2 n)$ 次。其总复杂度是 $O(n\log_2 n)$。

如果划分不是对称的，左部分和右部分的数量差别很大，那么复杂度会高一些。在极端情况下，例如左部分只有一个数，剩下的全部都在右部分，那么最多可能划分 n 次，总复杂度是 $O(n^2)$。所以，快速排序是不稳定的。

不过，一般情况下快速排序效率很高，甚至比稳定的归并排序更好。读者可以观察到，快速排序的代码比归并排序的代码简洁，代码中的比较、交换、复制操作很少。快速排序几乎是目前所有排序法中速度最快的方法。STL 的 sort()函数就是基于快速排序算法的，并针对快速排序的缺点做了很多优化。

视频讲解

快速排序思想可以用来解决一些特殊问题，例如求**第 k 大数**问题。

求第 k 大的数，简单的方法是用排序算法进行排序，然后定位第 k 大的数，其复杂度是 $O(n\log_2 n)$。

如果用快速排序的思想，可以在 $O(n)$ 的时间内找到第 k 大的数[①]。在快速排序程序中，每次划分的时候只要递归包含第 k 个数的那部分就行了。

【习题】

hdu 1425，求前 k 大的数；

poj 2388，求中间数。

6.3　减　治　法

大多数算法书籍不会特别讲解减治法(Decrease and Conquer)，减治法的题目常常被归纳到其他算法思想中。

用减治法解题的过程是把原问题分解为小问题，再把小问题分解为更小的问题，直到得到解。规模为 n 的原问题与分解后较小规模的子问题，它们的解有以下关系：

(1) 原问题的解只存在于其中一个子问题中；

(2) 原问题的解和其中一个子问题的解之间存在某种对应关系。

① 《算法导论》，Thomas H. Cormen，等著，潘金贵，等译，109 页，9.2 节。

按每次迭代中减去规模的大小把减治法分成以下 3 种情况：

(1) 减少一个常数。在算法的每次迭代中，把原问题减少相同的常数个，这个常数一般等于 1。相关的算法有插入排序、图搜索算法(DFS、BFS)、拓扑排序、生成排列、生成子集等。在这些问题中，每次把问题的规模减少 1。

(2) 按比例减少。在算法的每次迭代中，问题的规模按常数成倍减少，减少的效率极高。在大多数应用中，此常数因子等于 2。折半查找(Binary Search)是最典型的例子，在一个有序的数列中查找某个数 k，可以把数列分成相同长度的两半，然后在包含 k 的那部分继续折半，直到最后匹配到 k，总共只需要 $\log_2 n$ 次折半。

(3) 每次减少的规模都不同。减少的规模在算法的每次迭代中都不同，例如查找中位数(用快速排序的思路)、插值查找、欧几里得算法等。

6.4 小　　结

本章介绍了贪心、分治等基础算法的思想，这些也是算法竞赛中常见的题型。这两种算法思想容易理解、容易编程，若遇到难解的问题，大家不妨先考虑这两种方法。

第 7 章　动 态 规 划

✍动态规划的概念和思想
✍最优子结构和重叠子问题
✍基础 DP 和递推法
✍0/1 背包、LCS、LIS
✍滚动数组
✍记忆化搜索
✍区间 DP
✍树形 DP
✍数位 DP
✍状态压缩 DP

动态规划(Dynamic Programming,DP)题是算法竞赛中的必出题型。DP 算法的效率高、代码少,竞赛队员不仅需要掌握很多编程技术,而且需要根据题目灵活设计具体的解题方案,能考察其思维能力、建模抽象能力、灵活性等。对 DP 的掌握情况很能体现竞赛队员的思维水平。

本章详细展开了与 DP 有关的算法,这些算法是每个竞赛队员都应该掌握的基本技术。

和贪心法、分治法一样,DP 并不是一个特定的算法,而是一种算法思想。

DP 算法思想可以简单解释如下:DP 问题一般是多阶段决策问题,它把一个复杂问题分解为相对简单的子问题,再一个个解决,最后得到原复杂问题的最优解;这些子问题是前后相关的,并且非常相似,处理方法几乎一样。把前面子问题的计算结果记录为“状态”,并存储在“状态表”中,后面子问题可以直接查找前面得到的状态表,避免了重复计算,极大地减少了计算复杂度。

DP 和分治法的区别如下:

(1) 分治法是把问题分成独立的子问题,各个子问题能独立解决,一个子问题内部的计算不需要其他子问题的数据,例如归并排序的分治过程。

(2) DP 的子问题之间是相关的,前面子问题的解决结果被后面的子问题使用。

DP 比分治法复杂得多。

DP 适用于有重叠子问题和最优子结构性质的问题,具体的解释请参考算法类相关教材。

求解 DP 问题有 3 步,即定义状态、状态转移、算法实现。DP 的核心是状态、状态转移方程。用状态转移方程求解状态,**状态往往就是问题的解**。在 DP 问题中,只要分析出状态以及状态转移方程,差不多就完成了 90%的工作量。

DP 问题可以分为线性和非线性的。

(1) 线性 DP。线性 DP 有两种方法,即顺推与逆推。在线性 DP 中,常常用“表格”来处理状态,用表格这种图形化工具可以清晰易懂地演示推导过程。本章绘制了大量表格来介绍有关算法。

(2) 非线性 DP。例如树形 DP,建立在树上,也有两个方向:①根→叶,根传递有用的信息给子结点,最后根得出最优解;②叶→根,根的子结点传递有用的信息给根,最后根得到最优解。

DP 是一种常用的算法思想。DP 问题可难可易,非常灵活,重点在于对“状态”和“转移”的建模与分析。该算法时间效率高,代码量少。在几乎所有的现场赛中都有 DP 的影子,而且常常作为中等题、难题出现。DP 一直是算法竞赛中的重点和难点。

7.1 基础 DP

基础 DP 是一些经典问题,非常直观,易于理解。这些问题包括递推、0/1 背包、最长公共子序列、最长递增子序列等,它们的状态容易表示,转移方程容易得到。

下面从简单的硬币问题开始引导出动态规划的概念和处理方法。

7.1.1 硬币问题

前面第 6 章用贪心法解决的最少硬币问题要求硬币面值是特殊的。对于任意面值的硬币问题,需要用动态规划来解决。

硬币问题是简单的递推问题。

1. 最少硬币问题

有 n 种硬币,面值分别为 $v_1, v_2, \cdots, v_n$,数量无限。输入非负整数 s,选用硬币,使其和为 s。要求输出最少的硬币组合。

定义一个数组 int Min[MONEY],其中 Min[i]是金额 i 对应的最少硬币数量。如果程序能计算出 Min[i],$0<i<$MONEY,那么对输入的某个金额 i,只要查 Min[i]就得到了答案。

如何计算 Min[i]? Min[i]和 Min[$i-1$]是否有关系?

下面以 5 种面值(1、5、10、25、50)的硬币为例讲解递推的过程。

(1) 只使用最小面值的 1 分硬币。初始值 Min[0]=0,其他的 Min[i]为无穷大,如图 7.1 所示。下面计算 Min[1]。

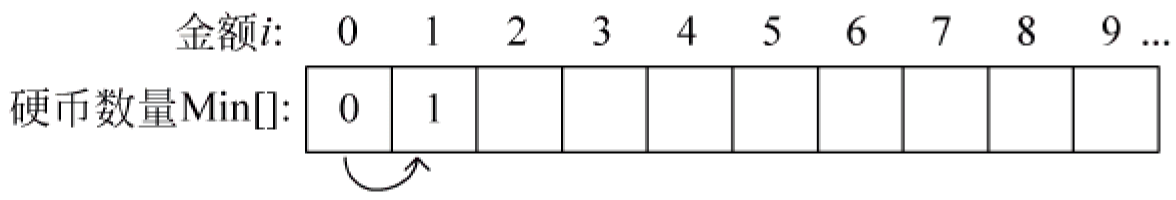


图 7.1　只用 1 分硬币

$i=0$,Min[0]=0,表示金额为 0,硬币数量为 0。在这个基础上加一个 1 分硬币,就前进到金额 $i=1$、硬币数量 Min[1]=Min[0]+1=Min[1-1]+1= 1 的情况。

同理，$i=2$ 时，相当于在 Min[1]的基础上加一个硬币，得到 Min[2]=Min[2−1]+1=2。继续这个过程，结果如图 7.2 所示。

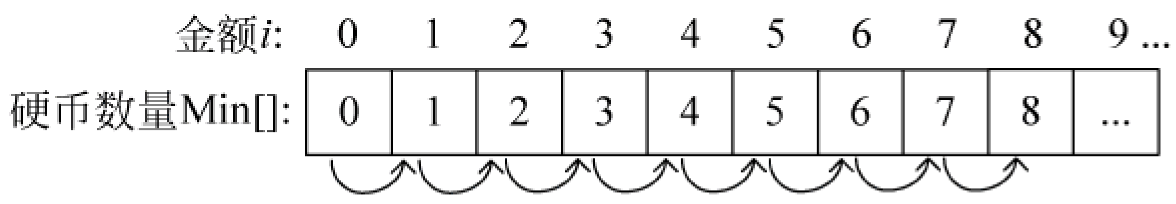


图 7.2 只用 1 分硬币时的结果

分析上述过程，得到递推关系 Min[i]=min(Min[i], Min[$i-1$]+1)。

(2) 在使用 1 分硬币的基础上增加使用第二大面值的 5 分硬币，如图 7.3 所示。此时应该从 Min[5]开始，因为比 5 分硬币小的金额不可能用 5 分硬币实现。

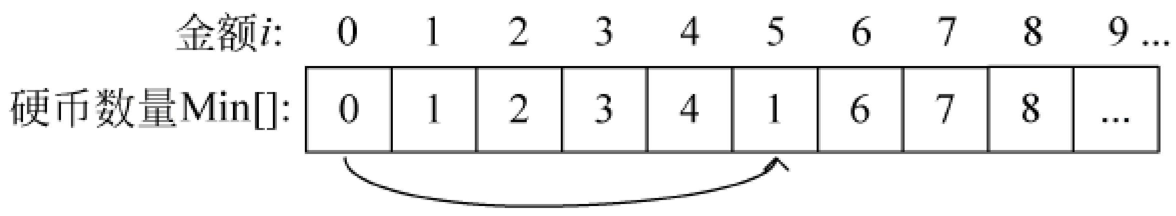


图 7.3 加上 5 分硬币

$i=5$ 时，相当于在 $i=0$ 的基础上加一个 5 分硬币，得到 Min[5]=Min[5−5]+1=1。上一步用 1 分硬币的方案有 Min[5]=5。取最小值，得 Min[5]=1。

同理，$i=6$ 时，有 Min[6]=Min[6−5]+1=2，对比原来的 Min[6]=6，取最小值。

继续这个过程，结果如图 7.4 所示。

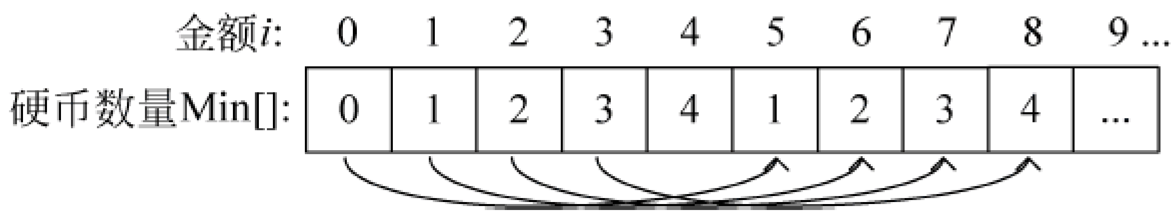


图 7.4 加上 5 分硬币时的结果

递推关系是 Min[i]=min(Min[i], Min[$i-5$]+1)。

(3) 继续处理其他面值的硬币。

在动态规划中，把 Min[i]这样的记录子问题最优解的数据称为“状态”，从 Min[$i-1$]或 Min[$i-5$]到 Min[i]的递推称为“状态转移”。用前面子问题的结果推导后续子问题的解，逻辑清晰、计算高效，这就是动态规划的特点。

程序代码如下：

```
#include <bits/stdc++.h>
using namespace std;
const int MONEY = 251;                    //定义最大金额
const int VALUE = 5;                      //5 种硬币
int type[VALUE] = {1, 5, 10, 25, 50};     //5 种面值
int Min[MONEY];                           //每个金额对应最少的硬币数量
void solve(){
    for(int k = 0; k < MONEY; k++)        //初始值为无穷大
        Min[k] = INT_MAX;
    Min[0] = 0;
    for(int j = 0; j < VALUE; j++)
        for(int i = type[j]; i < MONEY; i++)
```

```
            Min[i] = min(Min[i], Min[i - type[j]] + 1);     //递推式
}
int main(){
    int s;
    solve();                                //计算出所有金额对应的最少硬币数量,打表
    while(cin >> s)
        cout << Min[s] << endl;
    return 0;
}
```

solve()的复杂度是 $O(\text{VALUE}\times\text{MONEY})$。

需要注意的是,上面的 main()程序用到了"打表"的处理方法,即在输入金额之前提前用 solve()算出所有的解,得到 Min[MONEY]这个"表",然后再读取金额 s,查表直接输出结果,查一次表的复杂度只有 $O(1)$。这样做的原因是,如果有很多组测试数据,例如 10 000 个,那么总复杂度是 $O(\text{VALUE}\times\text{MONEY}+10\,000)$,没有增加多少。如果不打表,每次读一个 s,就用 solve()算一次,那么总复杂度是 $O(\text{VALUE}\times\text{MONEY}\times 10\,000)$,时间几乎多了 1 万倍。

2. 打印最少硬币的组合

在 DP 中,除求最优解的数量之外,往往还要求输出最优解本身,此时状态表需要适当扩展,以包含更多信息。

在最少硬币问题中,如果要求打印组合方案,需要增加一个记录表 Min_path[i],记录金额 i 需要的最后一个硬币。利用 Min_path[]逐步倒推,就能得到所有的硬币。

例如,金额 $i=6$,Min_path[6]=5,表示最后一个硬币是 5 分;然后,Min_path[6−5]=Min_path[1],查 Min_path[1]=1,表示接下来的最后一个硬币是 1 分;继续 Min_path[1−1]=0,不需要硬币了,结束。输出结果如图 7.5 所示,硬币组合是"5 分+1 分"。

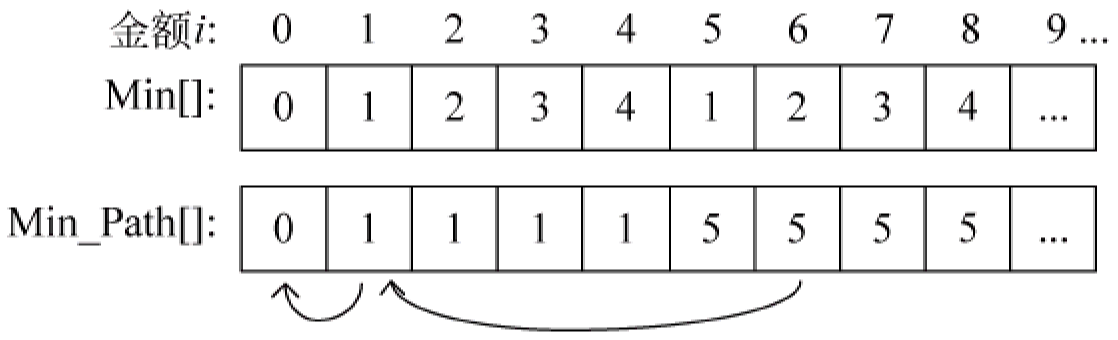


图 7.5 $i=6$ 时的输出结果

```
#include <bits/stdc++.h>
using namespace std;
const int MONEY = 251;                      //定义最大金额
const int VALUE = 5;                        //5 种硬币
int type[VALUE] = {1,5,10,25,50};           //5 种面值
int Min[MONEY];                             //每个金额对应最少的硬币数量
int Min_path[MONEY] = {0};                  //记录最小硬币的路径

void solve(){
    for(int k = 0; k < MONEY;k++)
        Min[k] = INT_MAX;
```

```
    Min[0] = 0;
    for(int j = 0;j < VALUE;j++)
        for(int i = type[j]; i < MONEY; i++)
            if(Min[i] >  Min[i - type[j]] + 1){
                Min_path[i] = type[j];          //在每个金额上记录路径,即某个硬币的面值
                Min[i] = Min[i - type[j]] + 1; //递推式
            }
}
void print_ans(int * Min_path, int s) {         //打印硬币组合
    while(s){
        cout << Min_path[s] << " ";
        s = s - Min_path[s];
    }
}
int main() {
    int s;
    solve();
    while(cin >> s){
        cout << Min[s] << endl;                  //输出最少硬币个数
        print_ans(Min_path,s);                   //打印硬币组合
    }
    return 0;
}
```

3. 所有硬币组合

有 n 种硬币,面值分别为 $v_1, v_2, \cdots, v_n$,数量无限。输入非负整数 s,选用硬币,使其和为 s。输出所有可能的硬币组合。

> **hdu 2069 "Coin Change"**
>
> 有 5 种面值的硬币,即 1 分、5 分、10 分、25 分、50 分。输入一个钱数 s,输出组合方案的数量。例如 11 分有 4 种组合方案,即 11 个 1 分、2 个 5 分+1 个 1 分、1 个 5 分+6 个 1 分、1 个 10 分+1 个 1 分。$s \leqslant 250$,硬币数量 num$\leqslant 100$。

如果用暴力法,可以逐个枚举各种面值的硬币个数,判断每种情况是否合法。枚举量是 $\frac{s}{50} \times \frac{s}{25} \times \frac{s}{10} \times \frac{s}{5} \times \frac{s}{1}$ 次。

1) 不完全解决方案

假设硬币数量不限,即题目没有 num$\leqslant 100$ 的限制。

定义一个记录状态的数组 int dp[251]。dp[i]表示金额 i 所对应的组合方案数,即解空间。找到 dp[i]和 dp[$i-1$]的递推关系,就高效地解决了问题。

第一步:只用 1 分硬币进行组合。

dp[0]=1 为初始值。

dp[1]可以从 dp[0]推导出来:当金额 $s=1$ 时,如果用一个 1 分硬币,等价于从 s 中减去 1 分钱,并且硬币数量也减少一个的情况。此时退到 $i=0$;如果 $i=0$ 存在组合方案,那么 $i=1$ 的组合方案也存在。dp[1]=dp[1]+dp[0]。

对于其他 dp[i]，同样有 dp[i]=dp[i]+dp[$i-1$]。

在上述叙述中，dp[i]是“状态”，dp[i]=dp[i]+dp[$i-1$]是状态转移方程。前面子问题的状态 dp[$i-1$]，用状态转移方程计算后，得到后面子问题的状态 dp[i]。

计算可得表 7.1。

表 7.1　只用 1 分硬币时

i	0	1	2	3	4	5	6	7	8	…
dp[i]	1	1	1	1	1	1	1	1	1	…

第二步：加上 5 分硬币，继续进行组合。

当 $i<5$ 时，组合中不可能有 5 分硬币。

当 $i\geqslant 5$ 时，金额为 s 时的组合数量等价于从 s 中减去 5，而且硬币数量也减去一个的情况。dp[i]=dp[i]+dp[$i-5$]。计算可得表 7.2。

表 7.2　加上 5 分硬币时

i	0	1	2	3	4	5	6	7	8	…
dp[i]	1	1	1	1	1	2	2	2	2	…

第三步：继续处理 10 分、25 分、50 分硬币的情况，同理有 dp[i]=dp[i]+dp[$i-10$]、dp[i]=dp[i]+dp[$i-25$]、dp[i]=dp[i]+dp[$i-50$]。

在上述步骤中，一次计算的复杂度只有 $O(1)$，全部计算的复杂度只有 $O(ks)$，k 是不同面值硬币的个数，s 是最大金额。

程序如下：

```
#include <bits/stdc++.h>
using namespace std;
const int MONEY = 251;                      //定义最大金额
int type[5] = {1, 5, 10, 25, 50};           //5 种面值
int dp[MONEY] = {0};
void solve() {
    dp[0] = 1;
    for(int i = 0;i < 5;i++)
        for(int j = type[i]; j < MONEY; j++)
            dp[j] = dp[j] + dp[j - type[i]];
}
int main() {
    int s;
    solve();                                //提前计算出所有金额对应的组合数量,打表
    while(cin >> s)
        cout << dp[s] << endl;
    return 0;
}
```

2）完全解决方案

上述程序有一个问题，没有考虑对硬币数量的限制，hdu 2069 题要求硬币不能多于 100 个。这是因为状态 dp[i]太简单，没有记录计算过程中的细节。

重新定义状态为 dp[i][j]，建立一个“转移矩阵”，如表 7.3 所示。其中，横向是金额（题目中 $i \leqslant 250$），纵向是硬币数（题目中最多用 100 个硬币，$j \leqslant 100$）。

表 7.3　转移矩阵

j \ i	0	1	2	3	4	5	6	7	8	9	10	…
0	1											
1		1				1						
2			1				1				1	
3				1				1				…
4					1				1			
5						1				1		
6							1				1	
7								1				…
…									…			
100												

矩阵元素 dp[i][j]的含义是用 j 个硬币实现金额 i 的方案数量。例如表 7.3 中 dp[6][2]=1，表示用两个硬币凑出 6 分钱，只有一种方案，即 5 分+1 分。该表中的空格为 0，即没有方案，例如 dp[6][1]=0，用一个硬币凑 6 分钱，不存在这样的方案。该表中列出了 dp[10][7]以内的方案数。

矩阵元素 dp[][]就是解空间。该表中纵坐标相加，就是某金额对应的方案总数，例如 6 分的金额为 dp[6][2]+dp[6][6]=2，有两种硬币组合方案。

“状态转移”的特征是用矩阵前面的状态 dp[i][j]能推算出后面状态的值。步骤如下：

第一步：只用 1 分硬币实现。

初始化：dp[0][0]=1，其他为 0。定义 int type[5]={1，5，10，25，50}为 5 种硬币的面值。

从 dp[0][0]开始，可以推导后面的状态。例如，dp[1][1]是 dp[0][0]进行“金额+1、硬币数量+1”后的状态转移。转移后组合方案数量不变，即 dp[1][1]=dp[0][0]=1。

这里还要考虑 dp[1][1]原有的方案数，递推关系修正为：

dp[1][1]=dp[1][1]+dp[0][0]=dp[1][1]+dp[1−1][1−1]=0+1=1

dp[1−1][1−1]的意思是从 1 分金额中减去 1 分硬币的钱，原来 1 个硬币的数量也减少 1 个。

在程序中，把上述操作写成：

dp[1][1]=dp[1][1]+dp[1−type[0]][1−1]

对所有 dp[i][j]进行上述操作，结果如表 7.4 所示。

表 7.4　只用 1 分硬币时

	0	1	2	3	4	5	6	7	8	9	10	…
0	1											
1		1										
2			1									
3				1								
4					1							
5						1						
6							1					
7								1				
…									…			
100												

第二步：加上 5 分硬币，继续进行组合。

dp[i][j]，当 $i<5$ 时，组合中不可能有 5 分硬币。

当 $i\geqslant 5$ 时，金额为 i、硬币为 j 个的组合数量等价于从 i 中减去 5 分钱，而且硬币数量也减去 1 个(即这个面值 5 的硬币)的情况。dp[i][j]＝dp[i][j]＋dp[$i-5$][$j-1$]＝dp[i][j]＋dp[$i-$type[1]][$j-1$]。对所有 dp[i][j]进行上述操作，结果如表 7.5 所示。

表 7.5　加上 5 分硬币时

	0	1	2	3	4	5	6	7	8	9	10	…
0	1											
1		1				1						
2			1				1				1	
3				1				1				…
4					1				1			
5						1				1		
6							1				1	
7								1				…
…									…			
100												

第三步：陆续加上 10 分、25 分、50 分硬币，同理有以下关系。

$$\text{dp}[i][j]=\text{dp}[i][j]+\text{dp}[i-\text{type}[k]][j-1],k=2,3,4$$

总结上述过程，每个状态 dp[i][j]都可以根据它前面已经算出的状态进行推导，复杂度为 $O(1)$，总复杂度为 $O(kmn)$，k 是不同面值硬币的个数，m 和 n 是矩阵的大小。

利用 dp[i][j]也很容易找到某金额对应的最少和最多硬币数量。例如金额 5，最少硬币数量是 dp[5][1]＝1，最多硬币数量是 dp[5][5]＝5。

下面给出代码。

hdu 2069 代码

```
#include<bits/stdc++.h>
using namespace std;
const int COIN = 101;                          //题目要求不超过 100 个硬币
const int MONEY = 251;                         //题目给定的钱数不超过 250
int dp[MONEY][COIN] = {0};                     //DP 转移矩阵
int type[5] = {1, 5, 10, 25, 50};              //5 种面值
void solve() {                                 //DP
    dp[0][0] = 1;
    for(int i = 0; i<5; i++)
        for(int j = 1; j<COIN; j++)
            for(int k = type[i]; k < MONEY; k++)
                dp[k][j] += dp[k - type[i]][j - 1];
}
int main() {
    int s;
    int ans[MONEY] = {0};
    solve();                                   //用 DP 计算完整的转移矩阵
    for(int i = 0; i< MONEY; i++)              //对每个金额计算有多少种组合方案,打表
        for(int j = 0; j<COIN; j++)            //从 0 开始,注意 dp[0][0] = 1
            ans[i] += dp[i][j];
    while(cin >> s)
        cout << ans[s] << endl;
    return 0;
}
```

7.1.2 0/1 背包

0/1 背包是最经典的 DP 问题,没有之一。

背包问题:有多个物品,重量不同、价值不同,以及一个容量有限的背包,选择一些物品装到背包中,问怎么装才能使装进背包的物品总价值最大。根据不同的限定条件,可以把背包问题分为很多种,常见的有下面两种:

(1) 如果每个物体可以切分,称为一般背包问题,用贪心法求最优解。比如吃自助餐,在饭量一定的情况下,怎么吃才能使吃到肚子里的最值钱?显然应该从最贵的食物吃起,吃完了最贵的再吃第 2 贵的,这就是贪心法。

(2) 如果每个物体不可分割,称为 0/1 背包问题。仍以吃自助餐为例,这次食物都是一份份的,每一份必须吃完。如果最贵的食物一份就超过了你的饭量,那只好放弃。这种问题无法用贪心法求最优解。

1. 0/1 背包问题

给定 n 种物品和一个背包,物品 i 的重量是 w_i、价值为 v_i,背包的总容量为 C。在装入背包的物品时对每种物品 i 只有两种选择,即装入背包和不装入背包(称为 0/1 背包)。如何选择装入背包的物品,使得装入背包中的物品的总价值最大?

设 x_i 表示物品 i 装入背包的情况，当 $x_i=0$ 时不装入背包，当 $x_i=1$ 时装入背包，有以下约束条件和目标函数。

约束条件：

$$\sum_{i=1}^{n} w_i x_i \leqslant C \quad x_i \in \{0,1\}, \quad 1 \leqslant i \leqslant n$$

目标函数：

$$\max \sum_{i=1}^{n} v_i x_i$$

视频讲解

2. 用 DP 求解 0/1 背包

后面的描述都基于这个例子：有 4 个物品，其重量分别是 2、3、6、5，价值分别为 6、3、5、4，背包的容量为 9。

引进一个 $(n+1)\times(C+1)$ 的二维表 dp[][]，可以把每个 dp[i][j]都看成一个背包，dp[i][j]表示把前 i 个物品装入容量为 j 的背包中获得的最大价值，i 是纵坐标，j 是横坐标。

填表按照只装第 1 个物品、只装前两个物品、只装前 3 个物品的顺序，直到装完，如图 7.6 所示。这是从小问题扩展到大问题的过程。

背包容量	→	0	1	2	3	4	5	6	7	8	9
不装→	0										
装第 1 个→	1										
装前 2 个→	2										
装前 3 个→	3										
装前 4 个→	4										

图 7.6 填表的过程

步骤 1：只装第 1 个物品。

由于物品 1 的重量是 2，所以背包容量小于 2 的都放不进去，得 dp[1][0]=dp[1][1]=0；物品 1 的重量等于背包容量，装进去，背包价值等于物品 1 的价值，dp[1][2]=6；容量大于 2 的背包，多余的容量用不到，所以价值和容量 2 的背包一样，如图 7.7 所示。

		0	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0	0	0
$w_1=2, v_1=6$	1	0	0	<u>6</u>	6	6	6	6	6	6	6

图 7.7 只装第 1 个物品

步骤 2：只装前两个物品。

如果物品 2 的重量比背包容量大，那么不能装物品 2，情况和只装第 1 个物品一样。

下面填 dp[2][3]。物品 2 的重量等于背包容量，那么可以装物品 2，也可以不装：

(1) 如果装物品 2(重量是 3)，那么相当于只把物品 1 装到(容量－3)的背包中，如图 7.8 所示。

(2) 如果不装物品 2，那么相当于只把物品 1 装到背包中，如图 7.9 所示。

取(1)和(2)的最大值，得 dp[2][3]=max{3,6}=6。

		0	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0	0	0
$w_1=2, v_1=6$	1	0	0	6	6	6	6	6	6	6	6
$w_2=3, v_1=3$	2	0	0	6	***3+0***						

图 7.8　装物品 2

		0	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0	0	0
$w_1=2, v_1=6$	1	0	0	6	6	6	6	6	6	6	6
$w_2=3, v_1=3$	2	*0*	*0*	6	***6***						

图 7.9　不装物品 2

后续步骤：继续以上过程，最后得到图 7.10（图中的箭头是几个例子）。

		0	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0	0	0
$w_1=2, v_1=6$	1	0	0	6	6	6	6	6	6	6	6
$w_2=3, v_1=3$	2	0	0	6	6	6	9	9	9	9	9
$w_3=6, v_1=5$	3	0	0	6	6	6	9	9	9	***11***	11
$w_4=5, v_1=4$	4	0	0	6	6	6	9	9	***10***	11	11

图 7.10　最终结果

最后的答案是 dp[4][9]，即把 4 个物品装到容量为 9 的背包，最大价值是 11。

其算法复杂度是 $O(nC)$。

3. 输出 0/1 背包方案

现在回头看具体装了哪些物品，需要倒过来观察：

dp[4][9]＝max{dp[3][4]＋4，dp[3][9]}＝dp[3][9]，说明没有装物品 4，用 $x_4=0$ 表示；

dp[3][9]＝max{dp[2][3]＋5，dp[2][9]}＝dp[2][3]＋5＝11，说明装了物品 3，$x_3=1$；

dp[2][3]＝max{dp[1][0]＋3，dp[1][3]}＝dp[1][3]，说明没有装物品 2，$x_2=0$；

dp[1][3]＝max{dp[0][1]＋6，dp[0][3]}＝dp[0][1]＋6＝6，说明装了物品 1，$x_1=1$。

图 7.11 中的实线箭头指出了方案的路径。

		0	1	2	3	4	5	6	7	8	9	
	0	0	0	0	0	0	0	0	0	0	0	
$w_1=2, v_1=6$	1	0	0	6	***6***	6	6	6	6	6	6	$x_1=1$
$w_2=3, v_1=3$	2	0	0	6	***6***	6	9	9	9	9	9	$x_2=0$
$w_3=6, v_1=5$	3	0	0	6	6	6	9	9	9	11	***11***	$x_3=1$
$w_4=5, v_1=4$	4	0	0	6	6	6	9	9	10	11	***11***	$x_4=0$

图 7.11　查看具体装了哪些物品

4. 例题

hdu 2602 "Bone Collector"

"骨头收集者"带着体积为 V 的背包去捡骨头，已知每个骨头的体积和价值，求能装进背包的最大价值。$N \leqslant 1000, V \leqslant 1000$。

输入：第 1 行是测试数量，第 2 行是骨头数量和背包体积，第 3 行是每个骨头的价值，第 4 行是每个骨头的体积。

1

5 10

1 2 3 4 5

5 4 3 2 1

输出：最大价值。

14

代码如下：

```
#include<bits/stdc++.h>
using namespace std;
struct BONE{
    int val;
    int vol;
}bone[1011];
int T,N,V;
int dp[1011][1011];
int ans(){
    memset(dp,0,sizeof(dp));
    for(int i = 1; i <= N; i++)
        for(int j = 0; j <= V; j++){
            if(bone[i].vol > j)          //第 i 个物品太大,装不了
              dp[i][j]  =  dp[i-1][j];
            else                         //第 i 个物品可以装
            dp[i][j] = max(dp[i-1][j],
                      dp[i-1][j-bone[i].vol] + bone[i].val);
        }
    return dp[N][V];
}
int main(){
    cin >> T;
    while(T--){
        cin >> N >> V;
        for(int i = 1;i <= N;i++)  cin >> bone[i].val;
        for(int i = 1;i <= N;i++)  cin >> bone[i].vol;
        cout << ans() << endl;
    }
    return 0;
}
```

作为练习，请读者自己加上打印方案的程序。

5. 滚动数组

在处理 dp[][]状态数组的时候有一个小技巧：把它变成一维的 dp[]，以节省空间。观察上面的二维表 dp[][]可以发现，每一行是从上面一行算出来的，只跟上面一行有关系，跟更前面的行没有关系。那么用新的一行覆盖原来的一行就可以了。

hdu 2602（滚动数组程序）

```
int dp[1011];                              //替换 int dp[1011][1011];
int ans(){
    memset(dp, 0, sizeof(dp));
    for(int i = 1; i <= N; i++)
        for(int j = V; j >= bone[i].vol; j -- )  //反过来循环
            dp[j] = max(dp[j],dp[j - bone[i].vol] + bone[i].val);
    return dp[V];
}
```

注意，j 应该反过来循环，即从后面往前面覆盖。请读者思考原因。

经过滚动数组的优化，空间复杂度从 $O(NV)$ 减少为 $O(V)$。

动态规划题经常会给出很大的 N、V，此时需要使用滚动数组，否则会 MLE。

滚动数组也有缺点，它覆盖了中间转移状态，只留下了最后的状态，所以损失了很多信息，导致无法输出背包的方案。

【习题】

滚动数组请练习：

hdu 1024 “Max Sum Plus Plus”；

hdu 4576 “Robot”；

hdu 5119 “Happy Matt Friends”。

7.1.3 最长公共子序列

一个给定序列的子序列是在该序列中删去若干元素后得到的序列。例如 $X=(A,B,C,B,D,A,B)$，X 的子序列有(A,B,C,B,A)、(A,B,D)、(B,C,D,B)等。子序列和子串是不同的概念，子串的元素在原序列中是连续的。

给定两个序列 X 和 Y，当另一序列 Z 既是 X 的子序列又是 Y 的子序列时，称 Z 是序列 X 和 Y 的公共子序列。最长公共子序列是长度最长的子序列。

最长公共子序列(Longest Common Subsequence，LCS)问题：给定两个序列 X 和 Y，找出 X 和 Y 的一个最长公共子序列。

用暴力法找最长公共子序列需要先找出 X 的所有子序列，然后验证是否为 Y 的子序列。如果 X 有 m 个元素，那么 X 有 2^m 个子序列，若 Y 有 n 个元素，总复杂度大于 $O(n2^m)$。

用动态规划求 LCS，复杂度是 $O(nm)$。

例如,序列 $X=(a,b,c,f,b,c)$、$Y=(a,b,f,c,a,b)$,如图 7.12 所示。

用 $L[i][j]$表示子序列 X_i 和 Y_j 的最长公共子序列的长度。

当 $X_i=Y_j$ 时,找出 X_{i-1} 和 Y_{j-1} 的最长公共子序列,然后在其尾部加上 X_i 即可得到 X 和 Y 的最长公共子序列。

当 $X_i \neq Y_j$ 时,求解两个子问题:①求 X_{i-1} 和 Y_j 的最长公共子序列;②求 X_i 和 Y_{j-1} 的最长公共子序列。然后取其中的最大值。

$$L[i][j]=\begin{cases} L[i-1][j-1]+1 & X_i=Y_j, i>0, j>0 \\ \max\{L[i][j-1], L[i-1][j]\} & X_i \neq Y_j, i>0, j>0 \end{cases}$$

下面举例说明前几个步骤。

步骤 1:求 $L[1][1]$。有 $X_1=Y_1$,得 $L[1][1]=L[0][0]+1=1$,如图 7.13 所示。

		Y=	a	b	f	c	a	b
		0	1	2	3	4	5	6
X=	0							
a	1							
b	2							
c	3							
f	4							
b	5							
c	6							

图 7.12　序列 X 和 Y

		Y=	**a**	b	f	c	a	b
		0	1	2	3	4	5	6
X=	0	0	0	0	0	0	0	0
a	1	0	**1**					

图 7.13　求 $L[1][1]$

步骤 2:求 $L[1][2]$。有 $X_1 \neq Y_2$,得 $L[1][2]=\max\{L[1][1], L[0][2]\}=1$,如图 7.14 所示。

后续步骤:继续以上过程,最后得到图 7.15,$L[6][6]$就是答案。

		Y=	a	**b**	f	c	a	b
		0	1	2	3	4	5	6
X=	0	0	0	0	0	0	0	0
a	1	0	1 →	**1** ↓				

图 7.14　求 $L[1][2]$

		Y=	a	b	f	c	a	b
		0	1	2	3	4	5	6
X=	0	0	0	0	0	0	0	0
a	1	0	1	1	1	1	1	1
b	2	0	1	2	2	2	2	2
c	3	0	1	2	2	3	3	3
f	4	0	1	2	3	3	3	3
b	5	0	1	2	3	3	3	4
c	6	0	1	2	3	4	4	4

图 7.15　最终结果

如果要输出方案,和 0/1 背包的输出方案一样,LCS 需要从后面倒推回去。

下面给出一个例题。

hdu 1159 "Common Subsequence"

求两个序列的最长公共子序列。

代码如下：

```
#include <bits/stdc++.h>
using namespace std;
int dp[1005][1005];
string str1, str2;
int LCS(){
    memset(dp, 0, sizeof(dp));
    for(int i = 1; i <= str1.length(); i++)
        for(int j = 1; j <= str2.length(); j++){
            if(str1[i - 1]  ==  str2[j - 1])
                dp[i][j]  =  dp[i - 1][j - 1]  +  1;
            else
                dp[i][j]  =  max(dp[i - 1][j], dp[i][j - 1]);
        }
    return dp[str1.length()][str2.length()];
}
int main(){
    while(cin >> str1 >> str2)
        cout << LCS() << endl;
    return 0;
}
```

读者可以练习输出方案，并改为滚动数组。

7.1.4　最长递增子序列

最长递增子序列(Longest Increasing Subsequence，LIS)问题：给定一个长度为 N 的数组，找出一个最长的单调递增子序列。例如一个长度为 7 的序列 $A=\{5,6,7,4,2,8,3\}$，它最长的单调递增子序列为$\{5,6,7,8\}$，长度为 4。

下面给出一个例题。

hdu 1257 “最少拦截系统”

某国有一种导弹拦截系统，这种导弹拦截系统有一个缺陷：虽然它的第 1 发炮弹能够到达任意高度，但是以后每一发炮弹都不能超过前一发的高度。某天，雷达捕捉到敌国的导弹来袭，请计算最少需要多少套拦截系统。

输入：导弹总个数，导弹依此飞来的高度。

输出：最少要配备多少套这种导弹拦截系统。

输入样例：

8 389 207 155 300 299 170 158 65

输出样例：

2

这一题可以用贪心法做。假设发射了很多高度无穷大的导弹，在读入第 1 个炮弹时，一个导弹下降来拦截。以后每读入一个新的炮弹，都由能拦截它的最低的那个导弹来拦截。

最后统计用于拦截的导弹的个数,也就是最少需要的拦截系统的套数(请读者思考能否用这个贪心思想求最长递增子序列)。

下面用 DP 来做。

这个题目的思维转换有一些难度。题目的意思是统计一个序列中的单调递减子序列最少有多少个。这和最长递增子序列 LIS 有什么关系呢?

假设已经有了求 LIS 的算法,读者可能想这么做:把序列反过来,就变成了求反序列的递增子序列;先求反序列的第 1 个 LIS,然后从原序列中去掉这个 LIS,再对剩下的求第 2 个 LIS,直到序列为空;这些 LIS 的数量就是题目的解。

但是,其实并不用这么麻烦,这个题目实际上等价于求原序列的 LIS,这是一道求 LIS 的**裸题**,下面解释原因。

模拟计算过程:①从第 1 个数开始,找一个最长的递减子序列,即第 1 个拦截系统 X,在样例中是{389,300,299,170,158,65},去掉这些数,序列中还剩下{207,155};②在剩下的序列中再找一个最长递减子序列,即第 2 个拦截系统 Y,是{207,155}。

在 Y 中,至少有一个数 a 大于 X 中的某个数,否则 a 比 X 的所有数都小,应该在 X 中。所以,从每个拦截系统中拿出一个数能构成一个递增子序列,即拦截系统的数量等于这个递增子序列的长度。如果这个递增子序列不是最长的,那么可以从某个拦截系统中拿出两个数 c、d,在拦截系统中 $c>d$,c 和 d 不是递增的,这与递增序列的要求矛盾。

有多种方法可以求 LIS。

(1) 方法 1:上一节刚讲解了最长公共子序列 LCS,读者也许能想到借助 LCS。首先对序列排序,得到 $A'=\{2,3,4,5,6,7,8\}$,那么 A 的 LIS 就是 A 和 A' 的 LCS。其复杂度是 $O(n^2)$。

(2) 方法 2:直接用 DP 求解 LIS。

定义状态 dp[i],表示以第 i 个数为结尾的最长递增子序列的长度,那么:

$\text{dp}[i]=\max\{0,\text{dp}[j]\}+1,\quad 0<j<i, A_j<A_i$

最后答案是 max{dp(i)}。

方法 2 的复杂度也是 $O(n^2)$,和方法 1 一样。

代码如下:

hdu 1257(DP 程序)

```
#include<bits/stdc++.h>
using namespace std;
const int MAXN = 10000;
int n,high[MAXN];
int LIS(){
    int ans = 1;
    int dp[MAXN];
    dp[1] = 1;
    for(int i = 2; i<=n; i++){
        int max = 0;
        for(int j=1; j<i; j++)
            if(dp[j]>max  &&  high[j]<high[i])
```

```
                max = dp[j];
        dp[i] = max + 1;
        if(dp[i] > ans) ans = dp[i];
    }
    return ans;
}
int main(){
    while(cin >> n){
        for(int i = 1; i <= n; i++)
            cin >> high[i];                    //输入炮弹高度值
        cout << LIS() << endl;
    }
    return 0;
}
```

(3) 方法 3：有一种更快的、复杂度为 $O(n\log_2 n)$的方法。这个方法不是 DP 算法，它巧妙地利用了序列本身的特征，通过一个辅助数组 $d[]$统计最长递增子序列的长度。

定义：数组 $d[]$；len 统计 $d[]$内数据的个数；high[]为原始序列。

初始化：$d[1]$=high[1]；len=1；

操作步骤：逐个处理 high[]中的数字，例如处理到了 high[k]，①如果 high[k]比 $d[]$末尾的数字更大，就加到 $d[]$的后面；②如果 high[k]比 $d[]$末尾的数字小，就替换 $d[]$中第 1 个大于它的数字。

以 high[]={4,8,9,5,6,7}为例，表 7.6 所示为具体的操作过程。

表 7.6　方法 3 的具体操作过程

i	high[]	$d[]$	len	说　　明
1	***4***, 8, 9, 5, 6, 7	***4***	1	初始值 $d[1]$=high[1]
2	4, ***8***, 9, 5, 6, 7	4 ***8***	2	high[2]>$d[1]$，加到 $d[]$的后面
3	4, 8, ***9***, 5, 6, 7	4 8 ***9***	3	$d[]$后面加上 9
4	4, 8, 9, ***5***, 6, 7	4 ***5*** 9	3	5 比 $d[]$末尾的 9 小，用 5 替换 $d[]$中第 1 个比 5 大的数 8
5	4, 8, 9, 5, ***6***, 7	4 5 ***6***	3	用 6 替换 9
6	4, 8, 9, 5, 6, ***7***	4 5 6 ***7***	4	$d[]$后面加上 7

结束后，len=4，就是 LIS 的长度。

为什么 $d[]$的长度等于 high[]的 LIS 的长度？分析算法对 high[]的两个关键操作：

(1) “如果 high[k]比 $d[]$末尾的数字更大，就加到 $d[]$后面”，high[]的 LIS 加 1，$d[]$的长度加 1，没有问题。

(2) “如果 high[k]比 $d[]$末尾的数字小，就替换 $d[]$中第 1 个大于它的数字”，有两个作用：首先，这个操作不影响 LIS 的长度，也不影响 $d[]$的长度；其次，high[]后面还没有处理的数很多都比已经处理过的数小，但是有可能序列更长，这里的替换给后面更小的数字留下了机会。为什么用 high[k]替换 $d[]$中第 1 个比它大的数字？因为数字 high[k]可能在 LIS 中，而被它替换的数字由于更大，不在 LIS 中的可能性更大。

在下面的代码中，对于“替换 $d[]$中第 1 个大于它的数字”这个功能，用 STL 的 lower_bound()函数帮助找到这个数，lower_bound()的复杂度是 $O(\log_2 n)$。程序的总复杂度是 $O(n\log_2 n)$[①]。

hdu 1257（非 DP 程序）

```
int LIS(){
    int len = 1;
    int d[MAXN];
    d[1] = high[1];                          //初始化
    for (int i = 2; i <= n; i++){            //O(n)
        if (high[i] > d[len])                //符合递增的要求，加入
            d[++len] = high[i];
        else{                                //替换
            int j = lower_bound(d + 1,d + len + 1,high[i]) - d;   //O(logn)
            d[j] = high[i];
        }
    }
    return len;
}
```

7.1.5 基础 DP 习题

(1) 简单题：

hdu 2018/2041/2044/2050/2182/4489。

(2) 背包：

有 0/1 背包、完全背包、分组背包、多重背包等。

hdu 1864 “最大报销额”，0/1 背包。

hdu 2159 “FATE”，完全背包。

hdu 2844 “Coins”，多重背包。

hdu 2955 “Robberies”，0/1 背包。

hdu 3092 “Least common multiple”，完全背包＋数论。

poj 1015 “Jury Compromise”。

poj 1170 “Shopping Offers”，状态压缩背包。

(3) LIS：

hdu 1003 “Max Sum”，最大连续子序列。

hdu 1087 “Super Jumping!”。

hdu 4352 “XHXJ's LIS”，数位 DP＋LIS。

poj 1239 “Increasing Sequence”，两次 dp。

(4) LCS：

hdu 1503 “Advanced Fruits”，LCS 变形。

① 最长不下降子序列是类似的问题。

poj 1080 “Human Gene Functions”，LCS 变形。

7.2　递推与记忆化搜索

前面讲解 DP 的状态转移都是用递推的方法，另外还有一种方法，逻辑上的理解更加直接，这就是用“递归＋记忆化搜索”来实现 DP。

先看一道经典题。

> **poj 1163 “The Triangle”**
>
> 给定一个 n 层的三角形数塔，从顶部第 1 个数往下走，每层经过一个数字，直到最底层。注意，只能走斜下方的左边一个数或右边一个数。问所有可能走到的路径，最大的数字和是多少？

此题如果按“从顶往下”的计算方法，则由于可能有 2^n 个路径，导致 TLE。请读者思考为什么会有 2^n 个路径。

更快的方法是“从底往上”计算。按动态规划的思路，对数塔上的每个点记录状态，dp[i][j]记录从第 i 层第 j 个数开始往下走的数字和，每个结点算一次，一共有 $O(n^2)$个结点，所以复杂度是 $O(n^2)$。计算过程如图 7.16 所示，括号内的数字是 dp[i][j]：

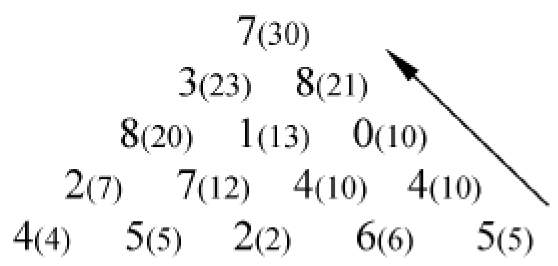


图 7.16　数塔

递推代码如下：

```
int a[150][150];                        //a[i][j]是数塔第 i 层的第 j 个数
int dp[150][150];                       //dp[i][j]记录从第 i 层第 j 个数开始往下走的数字和
for(int j = 1; j <= n;j++)  dp[n][j] = a[n][j];  //先计算最后一层
for(int i = n - 1;i >= 1;i -- )                  //从倒数第 2 层往上走到第 1 层
    for(int j = 1;j <= i;j++)                    //从左边走上来,或者从右边走上来,取其中较大的
        dp[i][j] = a[i][j] + max(dp[i + 1][j], dp[i + 1][j + 1]);
```

下面用“递归＋记忆化搜索”重新写 DP。

首先写出递归程序，搜索所有可能的路径。

视频讲解

```
int dfs(int i, int j) {
    if(i == n)
        return a[i][j];                 //递归边界:到达最后一行,返回
    return dp[i][j] = max(dfs(i + 1, j), dfs(i + 1, j + 1)) + a[i][j];
                                        //从左边走上来,或者从右边走上来,取其中较大的
}
```

这个 dfs()程序和前面“搜索技术”一章中讲解的 DFS 一样，是**暴力搜索**所有可能的情况。读者可以手工模拟递归过程，执行 dfs(1,1)，程序一直递归到最底部的第 n 层，然后逐步回退，最后回到最顶部的第 1 层。最后的结果在 dp[1][1]中。dfs()的递归有 2^n 次，**暴力**

搜索了所有的 2^n 个路径，复杂度和前面最先提到的“从顶往下”的计算次数一样。

这个递归程序能优化吗？可以观察到，其中有大量重复计算，其实是能避免的。例如，观察图 7.16 中第 3 层的中间数“1”，从第 2 层的“3”往下走会经过“1”，计算一次从“1”出发的递归；从第 2 层的“8”往下走也会经过“1”，又重新计算了从“1”出发的递归。所以，只要避免这些重复计算就能优化。

下面的代码加上了“记忆化搜索”的内容：

```
//把 dp[][]初始化为 -1
int dfs(int i, int j) {
    if(i == n)
        return a[i][j];
    if(dp[i][j] >= 0)                    //记忆!如果计算过，就不再递归重算
        return dp[i][j];
    return dp[i][j] = max(dfs(i+1,j), dfs(i+1,j+1)) + a[i][j];
}
```

其中“if(dp[i][j] >= 0) return dp[i][j];”实现了“记忆化搜索”。

加上这一行代码后，如果发现 dp[i][j]已经计算过，就不再重算。由于数塔的结点有 $O(n^2)$个，每个点只需要计算一次 dp[i][j]，所以 dfs()的运行次数只有 $O(n^2)$次，和递推程序的复杂度一样。这样，就把暴力搜索的 $O(2^n)$次计算优化到了 $O(n^2)$次计算。记忆化搜索的优化能力是惊人的。

记忆化搜索。在用递归实现 DP 时，在递归程序中记录计算过的状态，并在后续的计算中跳过已经算过的重复的状态，从而大大减少递归的计算次数，这就是“记忆化搜索”的思路。

在很多情况下，“记忆化搜索”的逻辑思路和程序比直接写递推更简单。在本书“7.5 数位 DP”中有相关的例子。

7.3 区　间　DP

区间 DP 的主要思想是先在小区间进行 DP 得到最优解，然后再利用小区间的最优解合并求大区间的最优解。

区间 DP，一般需要从小到大枚举所有可能的区间。在解题时，先解决小区间问题，然后合并小区间，得到更大的区间，直到解决最后的大区间问题。合并的操作一般是把左、右两个相邻的子区间合并。

区间 DP 的两个难点：枚举所有可能的区间、状态转移方程。

区间 DP 的复杂度：一个长度为 n 的区间，它的子区间数量级为 $O(n^2)$，每个子区间内部处理时间不确定，合起来复杂度会大于 $O(n^2)$。在编程时，区间 DP 至少需要两层 for 循环，第 1 层的 i 从区间的首部或尾部开始，第 2 层的 j 从 i 开始到结束，i 和 j 一起枚举出所

有的子区间。例如：

```
for(int i = 1;i < n;i++)                    //n 是区间长度
    for(int j = i;j <= n;j++)               //j 每次递增 1,也可能跨步递增
        …
```

下面用两个经典问题讲解。

1. 石子合并

> **“石子合并”**
>
> 有 n 堆石子排成一排,每堆石子有一定的数量,将 n 堆石子合并成一堆。合并的规则是每次只能合并相邻的两堆石子,合并的花费为这两堆石子的总数。石子经过 $n-1$ 次合并后成为一堆,求总的最小花费。
>
> 输入:有多组测试数据,输入到文件结束。每组测试数据的第 1 行有一个整数 n,表示有 n 堆石子,$n<250$。接下来的一行有 n 个数,分别表示这 n 堆石子的数目。每堆石子至少 1 颗,最多 10 000 颗。
>
> 输出:总的最小花费。
>
> 输入样例:
>
> 3
>
> 2 4 5
>
> 输出样例:
>
> 17

样例的计算过程是:①第一次合并 $2+4=6$;②第二次合并 $6+5=11$;总花费是 $6+11=17$。

DP 的状态如何设计?设 dp[i][j]为从第 i 堆石子到第 j 堆石子的最小花费,那么 dp[1][n]就是答案。另外,设 sum[i][j]为从第 i 到 j 的区间的和。

为了计算最后的 dp[1][n],需要考虑所有可能的合并。这些合并包括:

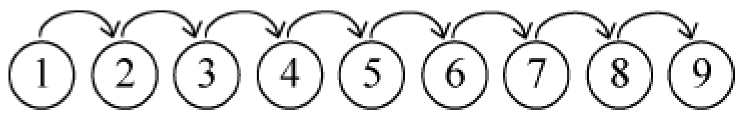


图 7.17　两堆合并

(1) 合并之前,dp[i][i]=0,$1\leqslant i\leqslant n$。

(2) 两堆合并,如图 7.17 所示。

例如:dp[1][2]=dp[1][1]+dp[2][2]+sum[1][2];

总结:dp[i][$i+1$]=dp[i][i]+dp[$i+1$][$i+1$]+sum[i][$i+1$];

(3) 三堆合并,如图 7.18 所示。

例如合并第 1 堆到第 3 堆,有两种情况,如图 7.19 所示。

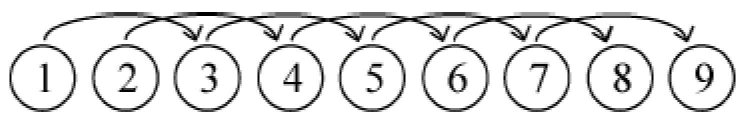


图 7.18　三堆合并

图 7.19　两种情况

dp[1][3]=min(dp[1][1]+dp[2][3],dp[1][2]+dp[3][3])+sum[1][3]；

总结：dp[i][i+2]=min(dp[i][i]+dp[i+1][i+2],dp[i][i+1]+dp[i+2][i+2])+sum[i][i+2]；

(4) 推广：第 i 堆到第 j 堆的合并，如图 7.20 所示。

dp[i][j] = min(dp[i][k] + dp[k + 1][j]) + sum[i][$j-i+1$]，$i \leqslant k \leqslant j$。这就是状态转移方程。

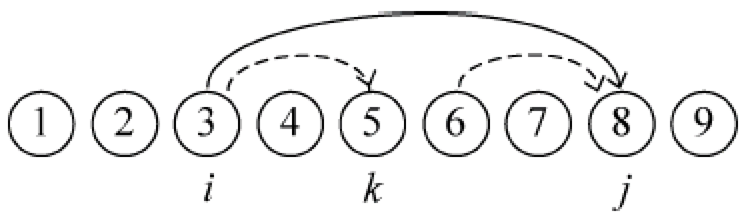


图 7.20　第 i 堆到第 j 堆的合并

下面的函数 Minval()实现了上述计算过程，其中有 3 层循环：

(1) 最外面一层的变量 len 表示区间[i,j]的长度，从 2 到 n；

(2) 第二层枚举的起点位置 i 从 1 到 n−len，终点通过计算得到，j=i+len；

(3) 在区间[i,j]里枚举每个分割的位置 k。

虽然下面的代码很短，但是逻辑比较复杂，请读者仔细体会并能自己写出来。

```
#include <bits/stdc++.h>
using namespace std;
const int INF = 1 << 30;
const int N = 300;
int sum[N], n;
int Minval() {
    int dp[N][N];
    for(int i = 1; i <= n; i++)
        dp[i][i] = 0;
    for(int len = 1; len < n; len++)                    //len 是 i 和 j 之间的距离
        for(int i = 1; i <= n - len; i++) {             //从第 i 堆开始
            int j = i + len;                            //到第 j 堆结束
            dp[i][j] = INF;
            for(int k = i; k < j; k++)                  //i 和 j 之间用 k 进行分割
                dp[i][j] = min(dp[i][j],
                      dp[i][k] + dp[k + 1][j] + sum[j] - sum[i - 1]);
        }
    return dp[1][n];
}
int main() {
    while(cin >> n) {
        sum[0] = 0;
        for(int i = 1; i <= n; i++) {
            int x;
            cin >> x;
            sum[i] = sum[i - 1] + x;                    //sum[i,j]的值等于 sum[j] - sum[i - 1]
        }
        cout << Minval() << endl;
    }
    return 0;
}
```

复杂度：Minval()中有三重循环，复杂度是 $O(n^3)$。所以上述算法只能用来处理规模 $n<250$ 的问题。

那么 Minval()是否可以优化？在它的三重循环中，前两重循环是枚举所有可能的合并，无法优化，最后一层循环枚举分割点 k，是可以优化的。因为每次运行最后一层循环时都在某个子区间内部寻找最优分割点，该操作在多个子区间里是重复的。如果找到这个最优点后保存下来，用于下一次循环，就能避免重复计算，从而降低复杂度。

用 $s[i][j]$表示区间$[i,j]$中的最优分割点，第三重循环可以从区间$[i,j-1)$的枚举优化到区间$[s[i][j-1],s[i+1][j]]$的枚举。其中，$s[][]$值是在前面的第三重循环中找到并记录下来的。

上述讨论符合“平行四边形优化”的原理，它是区间 DP 的常见优化方法。请读者自行了解并掌握。

经过优化以后，复杂度接近 $O(n^2)$，可以解决 $n<3000$ 的问题。上面的程序只需要修改 3 处，在下面的代码中使用斜体显示：

```
int Minval() {
    int dp[N][N], s[N][N];
    for(int i = 1; i <= n; i++){
        dp[i][i] = 0;
        s[i][i] = i;                                //初始值
    }
    for(int len = 1; len < n; len++)
        for(int i = 1; i <= n - len; i++)  {
            int j = i + len;
            dp[i][j] = INF;
            for(int k = s[i][j - 1]; k <= s[i + 1][j]; k++ )  //缩小范围
                if(dp[i][k] + dp[k + 1][j] + sum[j] - sum[i - 1]< dp[i][j]){
                    dp[i][j] = dp[i][k] + dp[k + 1][j] + sum[j] - sum[i - 1];
                    s[i][j] = k;                    //记录[i, j]的最优分割点
                }
        }
    return dp[1][n];
}
```

2. 回文串

回文串是正读和反读都一样的字符串，例如"abcdcba"。回文串问题是经典的字符串问题：给定一个字符串，然后通过增加或删除部分字符串得到一个回文串。

poj 3280 “Cheapest Palindrome”

给定字符串 s，长度为 m，由 n 个小写字母构成。在 s 的任意位置增删字母，把它变为回文串，增删特定字母的花费不同。求最小花费。

输入样例：

3 4

abcb

a 1000 1100

b 350 700

c 200 800

输出样例：

900

输入的第 1 行是 n 个字符，长度为 m；第 2 行是字符串 s，后面 n 行分别给出每个字符插入和删除的花费。

在样例中，如果在结尾处插入"a"，得到"abcba"，花费 1000；如果在首端删除"a"得到"bcb"，花费 1100；如果在首端插入"bcb"，花费 350＋200＋350＝900，这是最小值。

该题有多种解法，其中一种是把 s 反转得到 s'，求得两者最长公共子序列的长度 l，用 s 的长度减去 l 就是答案。

下面用区间 DP 的方法求解。

定义状态 dp[i][j]表示字符串 s 的子区间 s[i,j]变成回文的最小花费。

另外，在考虑删除和插入的花费时，由于这两种操作是等价的(这头加和那头减一样)，所以只要取这两种操作的最小值就行了。用数组 w[]定义字符的花费。

有以下 3 种情况：

(1) 如果 s[i]＝＝s[j]，那么 dp[i][j]＝dp[i＋1][j－1]，如图 7.21 所示。

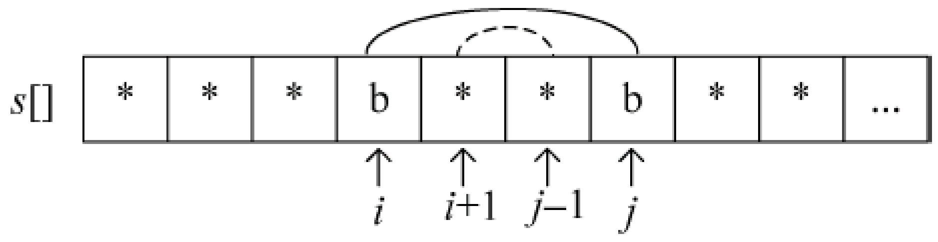


图 7.21　情况 1

(2) 如果 dp[i＋1][j]是回文串，那么 dp[i][j]＝dp[i＋1][j]＋w[i]，如图 7.22 所示。

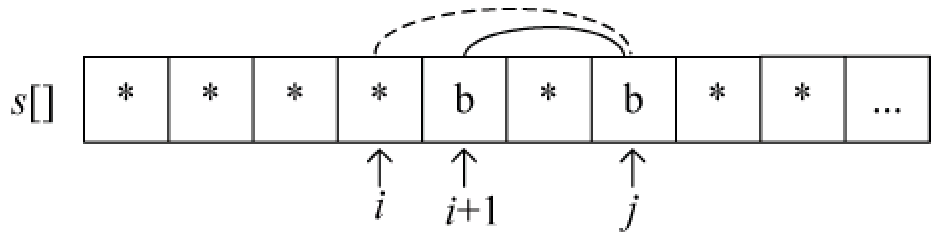


图 7.22　情况 2

(3) 如果 dp[i][j－1]是回文串，那么 dp[i][j]＝dp[i][j－1]＋w[j]。

总结情况 2、3，状态转移方程是 dp[i][j]＝min(dp[i＋1][j]＋w[i]，dp[i][j－1]＋w[j])。

该程序中包含两层循环，外层 i 枚举子区间起点，内层 j 枚举终点。因为需要从小区间扩展到大区间，所以 i 从 s 的尾端开始，逐步回退扩大区间，直到首端。

poj 3280 程序

```
#include <iostream>
using namespace std;
int w[30],n,m,dp[2005][2005];
char s[2005], ch;
int main() {
    int x,y;
    while(cin >> n >> m) {                          //n 是用到的字符个数,m 是 s 的长度
```

```
        cin >> s;
        for(int i = 0;i < n;i++) {
            cin >> ch >> x >> y;                    //读取每个字符的插入和删除的花费
            w[ch - 'a'] = min(x,y);                 //取其中的最小值
         }
         for(int i = m - 1; i >= 0; i-- )           //i 是子区间的起点
             for(int j = i + 1; j < m; j++) {       //j 是子区间的终点
                 if(s[i] == s[j])
                      dp[i][j] = dp[i + 1][j - 1];
                  else
                      dp[i][j] = min(dp[i + 1][j] + w[s[i] - 'a'],
                                 dp[i][j - 1] + w[s[j] - 'a']);
             }
             cout << dp[0][m - 1]<< endl;
    }
    return 0;
}
```

【习题】

hdu 3506 “Monkey Party”,环形石子合并。

hdu 4283 “You Are the One”,区间 DP。

hdu 4632 “Palindrome Subsequence”,回文串。

hdu 2476 “String Printer”,区间 DP。

hdu 4745 “Two Rabbits”,最长回文子序列。

hdu 5115 “Dire Wolf”,区间 DP。

poj 1141 “Brackets Sequence”,括号匹配。

poj 2955 “Brackets”,区间 DP。

7.4 树 形 DP

树形 DP 是指在“树”这种数据结构上进行的 DP:给出一棵树,要求以最少的代价(或取得最大收益)完成给定的操作。通常这类问题规模较大,枚举算法的效率低,无法胜任,贪心算法不能得到最优解,因此需要用动态规划。

在树上做动态规划显得非常合适,因为树本身有“子结构”性质(树和子树),具有递归性,符合 DP 的性质。相比线性 DP,树形 DP 的状态转移方程更加直观。不过,由于“树”这种数据结构比较烦琐,逻辑上比较复杂,状态转移方程不好设计,常常属于比较难的题目。

树的操作一般需要利用递归和搜索,用户需要熟练地掌握这些基础知识。树的遍历一般是从根结点往子结点方向深入,用 DFS 编程会比较简单。

下面从一个最基础的树形 DP 开始。

hdu 1520 "Anniversary Party"

一棵有根树上每个结点有一个权值，相邻的父结点和子结点只能选择一个，问如何选择使得总权值之和最大(邀请员工参加宴会，为了避免员工和直属上司发生尴尬，规定员工和直属上司不能同时出席)。

输入：结点编号从 1 到 N。第 1 行是一个数字 N，$1 \leqslant N \leqslant 6000$。后续 N 行中的每一行都包含结点的权值，范围是 -128 到 127 的整数。下面是 T 行，描述一个父子关系，每一行都有如下形式：

$L\ K$

第 K 个结点是第 L 个结点的父结点。读到 0 0 时结束。

输出：输出总的最大权值。

输入样例：

```
5
1
1
1
1
1
1 3
2 3
4 5
3 5
0 0
```

输出样例：

```
3
```

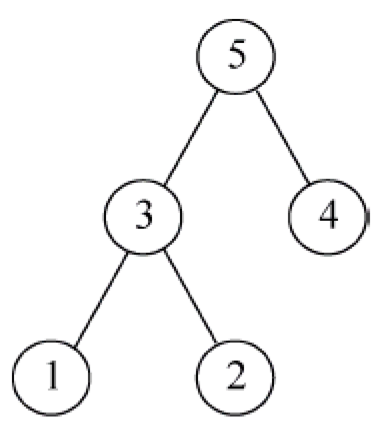


图 7.23　样例的树形关系

图 7.23 是样例的树结构。当结点选 1、2、5 时有最大值 3。读者可以思考，如果用暴力的方法遍历所有的情况，复杂度是多少。

根据 DP 的解题思路，定义状态为：

dp[i][0]，表示不选择当前结点时的最优解；

dp[i][1]，表示选择当前结点时的最优解。

状态转移方程有两种情况：

(1) 不选择当前结点，那么它的子结点可选可不选，取其中的最大值：

dp[u][0]+=max(dp[son][1], dp[son][0])

(2) 选择当前结点，那么它的子结点不能选，dp[u][1]+=dp[son][0]。

程序包含 3 个部分：

(1) 建树。本题可以用 STL 的 vector 生成链表，建立关系树。

(2) 树的遍历。可以用 DFS，从根结点开始进行记忆化搜索。

(3) DP。

```
#include <bits/stdc++.h>
using namespace std;
const int N = 6000 + 5;
int value[N], dp[N][2], father[N], n;
vector <int> tree[N];
void dfs(int u){
    dp[u][0] = 0;                                      //赋初值:不参加宴会
    dp[u][1] = value[u];                               //赋初值:参加宴会
    for(int i = 0;i < tree[u].size();i++){             //逐一处理这个父结点的每个子结点
        int son = tree[u][i];
        dfs(son);                                      //深搜子结点
        dp[u][0] += max(dp[son][1], dp[son][0]);
                                                       //父结点不选,子结点可选可不选
        dp[u][1] += dp[son][0];                        //父结点选择,子结点不选
    }
}
int main(){
    while(~scanf("%d",&n)) {
      for(int i = 1;i <= n;i++)  {
          scanf("%d",&value[i]);
          tree[i].clear();
          father[i] = -1;                              //赋初值,还未建立关系
      }
      while(1) {
        int a,b;
        scanf("%d%d",&a,&b);
        if(a == 0&&b == 0)  break;
        tree[b].push_back(a);                          //用邻接表建树
        father[a] = b;                                 //父子关系
      }
      int t = 1;
      while(father[t] != -1)                           //查找树的根结点
        t = father[t];
      dfs(t);                                          //从根结点开始,用 DFS 遍历整棵树
      printf("%d\n", max(dp[t][1], dp[t][0]));
    }
    return 0;
}
```

复杂度：上述代码遍历每个结点，总复杂度是 $O(n)$。

下面是一个中等难度的树形 DP 的题目，逻辑和状态转移都比较复杂。

hdu 2196 “Computer”

一棵有根树，根结点的编号是 1，对其中的任意一个结点，求离它最远的结点的距离。

输入：输入文件包含多个测试用例。每个用例的第 1 行是一个自然数 $N(N\leqslant 10\,000)$，后面有 $N-1$ 行。第 i 行包含两个自然数：某个结点；第 i 个结点连接到这个结点的距离，距离长度不超过 10 ^9。

输出：输出 N 行。第 i 行是距离第 i 个结点的最远距离。

输入样例：

5
1 1
2 1
3 1
1 1

输出样例：

3
2
3
4
4

复杂度分析：如果求从一个特定结点出发的最长路径，可以从这个结点出发，做一次 BFS，每次扩展邻居结点，并记录到这个邻居结点的最长距离，复杂度是 $O(n)$。求所有 n 个结点的最长距离，需要对每个结点单独做一次 BFS，总复杂度是 $O(n^2)$。但是由于题目规模较大，$N\leqslant 10\,000$，所以算法的复杂度最多只能是 $O(n\log_2 n)$。下面用动态规划求解。

一棵有根树如图 7.24 所示。

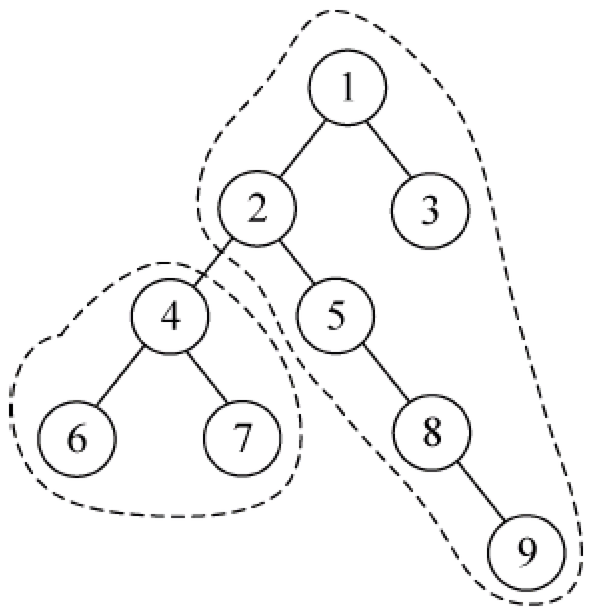


图 7.24　一棵有根树

以结点 4 为例，它的最长距离分两种情况讨论：

(1) 以 4 为顶点的子树(图 7.24 左边圈起的部分)距结点 4 的最远距离 L_1。对结点 4 来说，它的 L_1 很容易求，只要从结点 4 出发对它的子树做一次 DFS，记录最大深度，就能求得 L_1。那么如何求得树上所有结点的 L_1 值？可以从根结点 1 开始 DFS，遍历所有的结点，在 DFS 返回的过程中记录每个结点的最大深度，即这个结点的 L_1 值(在下面的程序中实际上计算了每个结点的两个距离。以结点 4 为例，这两个距离是以 4 为顶点的最长距离 one，即 L_1 值；以 4 为顶点的第 2 长距离 two)。

(2) 剩下部分(图 7.24 右边圈起的部分)到结点 4 的最长距离 L_2。L_2 = 父结点 2 的最长距离 + dist(2,4)，dist(2,4)是 2 和 4 之间的距离。求 L_2 的关键是求父结点 2 的最长距离，它又分两种情况：

① 从结点 2 往上走的最长距离，图中路径是 2-1-3。这可以通过 DFS 不断更新结点来获得，具体操作见下面的程序。

② 结点 2 除了结点 4 以外的其他子树的最长距离 X，图中路径是 2-5-8-9。在上面第

(1)步中已经求得了每个结点的最长距离 one 和次长距离 two。如果结点 4 在父结点 2 的最长子树上，那么 X＝two＋dist(2,4)；如果结点 4 不在父结点 2 的最长子树上，那么 X＝one＋dist(2,4)。

综上所述，距离结点 4 最远的距离是 max{L_1,L_2}。在程序中，用 dfs1()实现功能(1)，用 dfs2()实现功能(2)。

状态的设计：结点 i 的子树到 i 的最长距离 dp[i][0]以及次长距离 dp[i][1]；从结点 i 往上走的最短距离 dp[i][2]。

```
#include <bits/stdc++.h>
using namespace std;
const int N = 10100;
struct Node{
    int id;                                   //子结点的编号
    int cost;
};
vector<Node> tree[N];
int dp[N][3];
int n;
void init_read(){
    for(int i=1; i<=n; i++)
        tree[i].clear();
    memset(dp, 0, sizeof(dp));
    for(int i=2; i<=n; i++) {
        int x,y;
        scanf("%d%d",&x,&y);
        Node tmp;
        tmp.cost = y;
        tmp.id=i;                             //i 是 x 的子结点
        tree[x].push_back(tmp);
    }
}
void dfs1(int father) {                       //DFS,先处理子结点,再处理父结点
    int one=0, two=0;
    for(int i=0; i<tree[father].size(); i++) {
                                              //遍历结点 father 的所有子结点
        Node child = tree[father][i];
        dfs1(child.id);                       //递归子结点,直到最底层
        int cost = dp[child.id][0] + child.cost;
        if(cost >= one) {                     //用 one 记录从 father 往下走的最长距离
            two = one;                        //原来的最长距离 one 变成第 2 长,用 two 记录
            one = cost;
        }
        if(cost < one && cost > two)          //用 two 记录第 2 长的距离
            two = cost;
    }
    dp[father][0] = one;                      //得到以 father 为起点的子树的最长距离
    dp[father][1] = two;                      //得到以 father 为起点的子树的第 2 长距离
}
void dfs2(int father) {                       //先处理父结点,再处理子结点
```

```
    for(int i = 0; i < tree[father].size(); i++) {
        Node child = tree[father][i];
        if(dp[child.id][0] + child.cost == dp[father][0])
                                //child 在最长距离的子树上
            dp[child.id][2] = max(dp[father][2], dp[father][1]) + child.cost;
        else                              //child 不在最长距离的子树上
            dp[child.id][2] = max(dp[father][2], dp[father][0]) + child.cost;
        dfs2(child.id);
    }
}
int main(){
    while(~scanf("%d", &n)) {
        init_read();                      //初始化,读数据
        dfs1(1);                          //计算 dp[ ][0]、dp[ ][1]
        dp[1][2] = 0;
        dfs2(1);                          //计算 dp[ ][2]
        for(int i = 1; i <= n; i++)
            printf("%d\n", max(dp[i][0], dp[i][2]));
    }
    return 0;
}
```

复杂度：dfs1()和 dfs2()的复杂度约为 $O(n)$。

【习题】

下面题目的难度都在中等以上。

poj 2378/3107/3140；

hdu 1011/1561/2242/3586/5834。

7.5 数 位 DP

先从一道简单题引出数位 DP 的概念。

> **hdu 2089 "不要 62"**
>
> 一个数字，如果包含'4'或者'62'，它是不吉利的。给定 m 和 n，$0<m\leqslant n<10^6$，统计 $[m, n]$范围内吉利数的个数。

这一题的数据范围是 10^6，但是此类题目常常达到 10^{18}。暴力方法是检查每一个数，复杂度大于 $O(n)$。由于 n 太大，肯定会超时，需要设计一个时间复杂度接近 $O(\log_2 n)$的算法。

读者很容易想到排除法。基本思路是在 $0\sim10^6$ 内排除不符合条件的数，具体操作是按"从高位到低位"的顺序进行判断。例如，求 1～999 999 内不包含 4 的数(对数字'62'的处理方法类似)，步骤如下：

(1) 在 6 位数中排除最高位是 4 的数，即 400 000～499 999。虽然有 10 万个数，但只需要判断最高位，一次就全排除了。

(2) 在最高位不是 4 的 6 位数中排除次高位是 4 的数。例如最高位是 1 的数可以一次性排除 140 000～149 999，共 1 万个。注意，首位**可以是 0**，即 000 000～099 999 也算 6 位数。

(3) 继续排除 5 位数、4 位数等，直到结束。

下面用数位 DP 的方法实现上述排除法的思路。

所谓“数位 DP”，是指对数字的“位”进行的与计数有关的 DP。一个数有个位、十位、百位、千位等，数的每一位就是数位。数位 DP 用来解决与数字操作有关的问题，例如数位之和问题、特定数字问题等。这些问题的特征是给定的区间超级大，不能用暴力的方法逐个检查，必须用接近 $O(\log_2 n)$ 复杂度的算法。解题的思路是用 DP 对“数位”进行操作，记录已经算过的区间的状态，用在后续计算中，快速进行大范围的筛选。

回头考虑 hdu 2089 题，统计所有的吉利数，用 DP 怎么做？下面用两种方法实现 DP，一种用递推公式，一种用记忆化搜索。

1. 用递推实现 hdu 2089 题

定义状态 dp[i][j]，它表示 i 位数中首位是 j，符合要求的数的个数。例如 dp[6][1]表示首位是 1 的 6 位数，即 100 000～199 999 中符合要求的数有多少个。那么如何求 dp[6][1]？计算首位数字 1 后面的 5 位数就可以了，即计算 00 000～99 999 中符合要求的数。所以，dp[i][j]的递推公式如下：

$$\mathrm{dp}[i][j] = \sum_{k=0}^{9} \mathrm{dp}[i-1][k], \quad (j \neq 4)\&\&(k \neq 2\&\&j \neq 6)$$

下面是程序。为了突出对数位 DP 思路的理解，程序简化了题目的要求，只排除了'4'，没有排除"62"。作为练习，读者可以自己加上对"62"的处理。从下面的程序可知，求 dp[i][j]的计算复杂度极小，i、j、k 的三重循环只需要计算 1000 次。

统计[0,n]内不含 4 的数字个数(递推程序)

```
#include<bits/stdc++.h>
const int LEN = 12;                //可以更大
int dp[LEN][10];                   //dp[i][j]表示 i 位数,第 1 个数是 j 时符合条件的数字数量
int digit[LEN];                    //digit[i]存第 i 位数字
void init(){
    dp[0][0] = 1;
    for(int i = 1; i <= LEN; i++)
        for(int j = 0; j < 10; j++)
            for(int k = 0; k < 10; k++)
                if(j!= 4)              //排除数字 4
                    dp[i][j] += dp[i - 1][k];
}
int solve(int len) {                       //计算[0,n]区间满足条件的数字个数
    int ans = 0;
    for(int i = len; i >= 1; i-- ){        //从高位到低位处理
        for(int j = 0; j < digit[i]; j++)
            if(j!= 4)
```

```
                ans += dp[i][j];
            if(digit[i] == 4) {           //第 i 位是 4,以 4 开头的数都不行
              ans -- ; break;}
        }
        return ans;
    }
    int main(){
        int n,len = 0;
        init();                           //预计算 dp[][]
        scanf("%d",&n);
        while(n){                         //len 是 n 的位数.例如 n = 324,是 3 位数,len = 3
            digit[++len] = n%10;
                                          //例如 n = 324,digit[3] = 3, digit[2] = 2, digit[1] = 4
            n/ = 10;
        }
        printf("%d\n",solve(len) + 1);    //求[0,n]内不含 4 的数字个数
        return 0;
    }
```

程序中的 init()是预处理,求 dp[][],如表 7.7 所示(这里只画了部分箭头)。

表 7.7　dp[*i*][*j*]的计算

j \ i	0	1	2	3	4	5	6	7	8	9	10	…
0	1	1	9	81	729	…						
1		1	9	81	729	…						
2		1	9	81	729	…						
3		1	9	81	729	…						…
4		0	0	0	0	…						
5		1	9	81	729	…						
6		1	9	81	729	…						
7		1	9	81	729	…						…
8		1	9	81	729	…						
9		1	9	81	729	…						

然后用 solve()完成题目的计算。题目要求计算给定范围内符合要求的数,那么把相应的 dp[*i*][*j*]相加即可(加的时候需要判断'4')。例如,求[0, 324]内符合条件的数,设答案是 ans,计算步骤如下:

(1) 处理 3 位数,ans=ans+dp[3][0]+dp[3][1]+dp[3][2],得到 000～099、100～199、200～299 内符合条件的个数。

(2) 处理 2 位数,ans=ans+dp[2][0]+dp[2][1],得到 00～09、10～19 内符合条件的个数。实际上,这一步的计算对应的是 300～309、310～319。

(3) 处理 1 位数,即 ans=ans+dp[1][0]+dp[1][1]+dp[1][2]+dp[1][3]。实际上,这一步计算的是 320、321、322、323、324。

下面用记忆化搜索方法重新实现上述思路。

2. 用记忆化搜索实现 hdu 2089 题

回顾记忆化搜索，其思路是在递归程序 dfs()中搜索所有可能的情况，遇到已经算过的记录在 dp[]中的结果就直接使用，不再重复计算。

例如求[0，324]内符合条件的数，记忆化搜索的过程如图 7.25 所示。其中画线部分是前面已经算过的，记录在 dp[]中，不用再递归和重算。

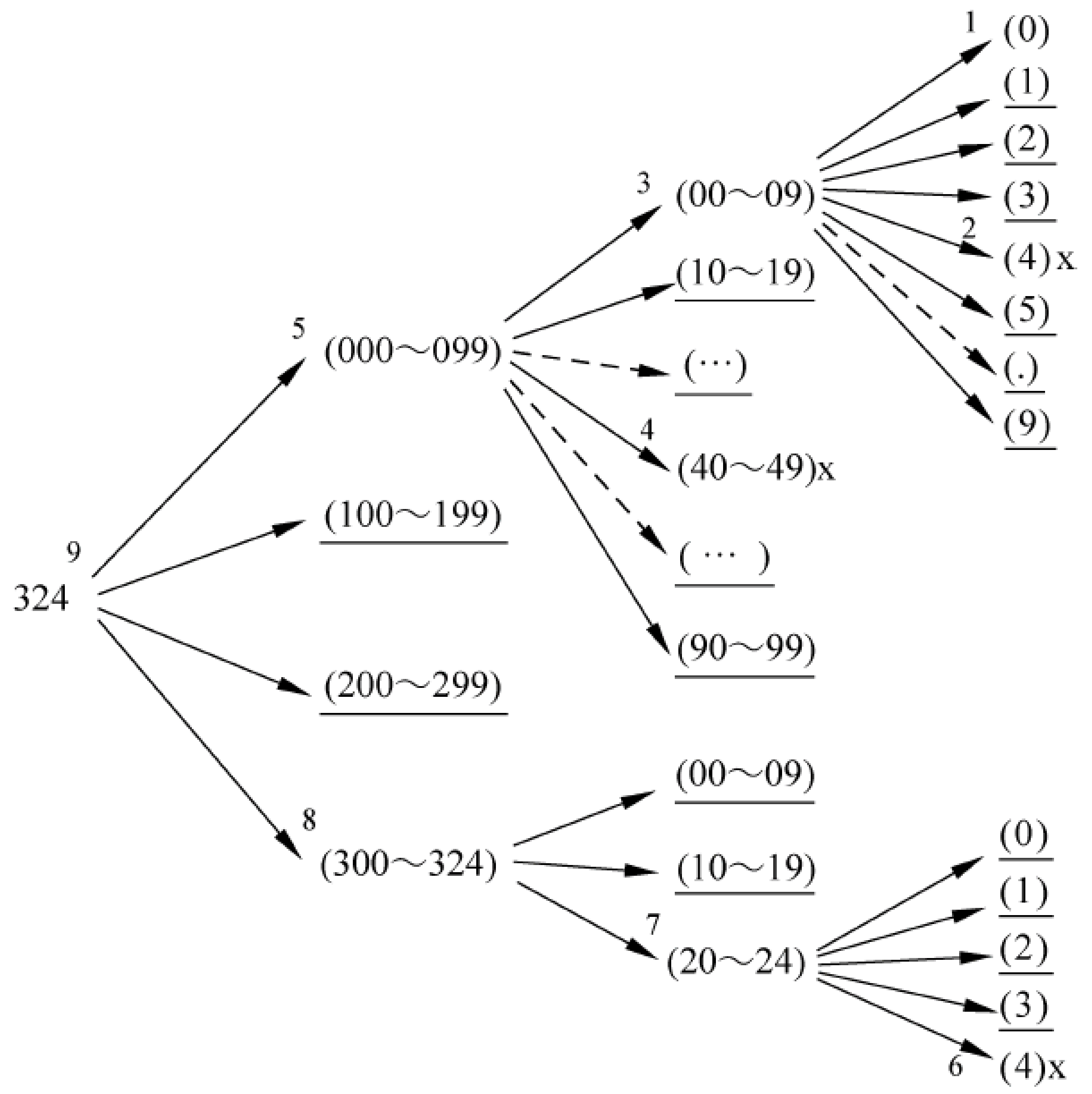


图 7.25　[0，324]的记忆化搜索过程

定义 dp[i]是 i 位数中符合要求的数字个数。dp[1]表示符合条件的 1 位数，0 是 1 位数，它的 dp[1]=1；1 也是 1 位数，它的 dp[1]沿用 0 算过的 dp[1]即可。dp[2]表示符合条件的 2 位数的个数，00～09 是 2 位数，计算得到 dp[2]=9；在搜索 10～19 时，沿用 dp[2]即可。同理，(100～199)和(200～299)都沿用(000～099)的计算结果 dp[3]，不用再计算。

dfs()的执行过程如下：从输入 324 开始，一直递归到最深处的(0)，然后逐步回退，计算的顺序是(0)→(4)→(00～09)→(40～49)→(000～099)→(4)→(20～24)→(300～324)→324，图 7.25 中用小写数字标识了这个顺序。

记忆化搜索极大地减少了搜索次数。例如图 7.25 中检查(000～099)，因为用 dp[]进行记忆化搜索，只需要计算 5 次；如果去掉记忆化部分，需要递归检查每个数，共 100 次。

下面是程序，程序只排除了数字'4'，读者自己练习排除"62"。对"62"的处理比较复杂，需要设计新的 dp[]状态。

```
#include <bits/stdc++.h>
const int LEN = 12;
int dp[LEN];            //dp[i]:i 位数符合要求的个数.例如 dp[2]表示 00～99 内符合要求的个数
int digit[LEN];
int dfs(int len, int ismax) {
    int ans = 0, maxx;
```

```
    if(!len) return 1;                  //已经递归到 0 位数,返回
    if(!ismax && dp[len]!= -1)          //记忆化搜索:如果已经算过,直接使用
        return dp[len];
    maxx = (ismax ? digit[len] : 9);
    for(int i = 0; i <= maxx; i++) {
        if(i == 4) continue;            //排除 4
        ans += dfs(len - 1, ismax && i == maxx);
    }
    if(!ismax)   dp[len] = ans;
    return ans;
}
int main(){
    int n, len = 0;
    memset(dp, -1, sizeof(dp));         //初始化为 -1
    scanf("%d", &n);
    while(n) {
        digit[++len] = n%10;
        n/ = 10;
    }
    printf("%d\n", dfs(len,1));
    return 0;
}
```

【习题】

hdu 3555 题：求[1,N]里面有多少数包含'49'，$1\leqslant N\leqslant 2^{63}-1$。

hdu 3652 题：B-number 是一个非负整数，其十进制形式包含'13'并且可以被 13 整除。给定整数 n，$1\leqslant n\leqslant 10^9$，计算 $1\sim n$ 的 B-number 数。

hdu 6148 题：计算不大于 N 的 Valley Number 个数，结果对 10^9+7 取模。此题较难。

hdu 4507 题：计算[L,R]中和 7 无关的数字的平方和，结果对 10^9+7 取模。$1\leqslant L\leqslant R\leqslant 10^{18}$。此题较难。

7.6　状态压缩 DP

先用一道例题引出状态压缩 DP 的概念。

1. 例题 1

poj 3254 “Corn Fields”

农夫约翰有一片长方形土地，划成 M 行 N 列的方格。他准备种玉米、养牛，不过有些格子很贫瘠，不适合种玉米。还有，牛不喜欢在一起吃，所以牛不能放在相邻的格子里。给出这块地的情况，求约翰有多少个种玉米的方案。所有方格都不种玉米也算一种方案。

输入：第 1 行是 M 和 N，$1 \leqslant M, N \leqslant 12$。后面有 M 行，描述方格情况，1 表示肥沃，0 表示贫瘠。

输出：方案数，用 10^9 取模。

输入样例：

2 3
1 1 1
0 1 0

输出样例：

9

提示：在样例中有 9 种方案。

1	2	3
	4	

分别是{}、{1}、{2}、{3}、{4}、{1,3}、{1,4}、{3,4}、{1,3,4}。

这个方格图共 $m \times n$ 个格子，有 $2^{m \times n}$ 种排列，无法用暴力法计算。

用下面的方法编程计算，算法复杂度是 $O(m2^n2^n)$。

1）方格的表示

很容易想到，可以用二进制数来描述方格，1 表示种玉米，0 表示不种玉米。在样例中，第 1 行的 3 个方格都是肥沃的，排除相邻的情况，有以下 5 种种玉米的方案：

第 1 行	编　号	1	2	3	4	5
	方　案	000	001	010	100	101

这里的编号并不是多余的，在下面设计 DP 状态的时候有用。

2）DP 状态和状态转移

如何设计 DP 状态，把问题从小规模逐步扩展到大规模？可以按行进行扩展。

上面已经得到了第 1 行的 5 种方案，下面继续扩展第 2 行。

在样例中，第 2 行只有两种方案，即 000、010。

第 2 行	编　号	1	2
	方　案	000	010

如果第 2 行选编号 1 的 000，第 1 行可以选 5 种方案而不冲突。

如果第 2 行选编号 2 的 010，与第 1 行的 010 有冲突，第 1 行的其他 4 种方案没问题。

共 $5+4=9$ 种方案。

用 dp[i][j]表示第 i 行采用第 j 种编号的方案时前 i 行可以得到的可行方案总数。例如，dp[2][2]=4 表示第 2 行使用第 2 种方案（即 010）时的方案总数是 4。

从第 $i-1$ 行转移到第 i 行，状态转移方程如下：

$$\mathrm{dp}[i][k]=\sum_{j=1}^{n}\mathrm{dp}[i-1][j]$$

其中 j 是第 $i-1$ 行可行方案的编号，而且所有的 $\mathrm{dp}[i-1][j]$ 与第 i 行不冲突。

把最后一行的 $\mathrm{dp}[m][k](1\leqslant k\leqslant n)$ 相加就得到了答案。

3）一些细节

程序有很多细节，例如初始化每一行的合法方案，即找没有相邻 1 的二进制数。用 state[]表示方案，例如样例的第 1 行 state[2]$=010_2$ 表示只种中间一块地。可以这样写程序：

```
int state[600];                          //state[x]:编号 x 的方案是 state[x]
bool check(int x){                       //判断 x 的二进制数是否有相邻的 1
    if(x&x << 1)return false;            //x 有相邻的 1,该方案不合法
    else      return true;               //x 没有相邻的 1,合法
}
void init(){                             //初始化合法的方案
    int j = 0;
    int total = 1 << N;                  //一行有 N 个格子,有 2^N 种情况
    for(int i = 0; i < total; ++i)
        if(check(i))   state[++j] = i;   //记录合法方案
}
```

对于相邻两行的合法性判断，这样写程序：

```
if(state[i] & state[j] != 1) ...         //相邻的两行,没有挨着的 1
```

2. 状态压缩 DP 的概念

从上面的例子可以看出，每个状态 $\mathrm{dp}[i][j]$ 表示的不是一个有意义的数值，例如前面章节中的花费、价值、长度等，而是代表了集合的数量。这种处理复杂集合问题的 DP 叫做状态压缩 DP。

集合的状态有很多，操作复杂，往往把方案用二进制数（“压缩”到这个二进制数中）来表示和操作。二进制操作有与、或、取反、移位等。在上面的例子中，把可能的方案“压缩”到 state[]中，操作用到了左移。

3. 旅行商问题

视频讲解

旅行商问题（Traveling Salesman Problem，TSP）是一个经典问题：有 n 个城市，已知任何两个城市之间的距离（或者费用），一个旅行商从某城市出发，经过每一个城市并且只经过一次，最后回到出发的城市，输出最短（或者路费最少）的线路。

TSP 问题是 NP 难度的，没有多项式时间的高效算法，所以 TSP 题目给的 n 都很小。如果用暴力法，可以列出所有的路线，然后逐一判断。路线最多可能有 $(n-1)!$ 种，只能用于解决规模 $n\leqslant 11$ 的问题。如果题目不需要求最短的线路，可以用贪心法求近似解，找出一条可行的、比较短的路线。

小规模的 TSP 问题可以用状态压缩 DP 求解，复杂度是 $O(2^n n^2)$，能解决规模 $n\leqslant 15$ 的问题，比暴力的 $O(n!)$ 好一些。思路如下：

假设最短的 TSP 路径是 Path$=(v_0\to v_1\to v_2\to v_3\to v_0)$

那么 Path=$(v_0 \to v_1)+(v_1 \to v_2 \to v_3 \to v_0)$

所以,问题转变为:求经过所有城市的最短回路→从某个城市到起点的最短路径。

DP 状态设计如下:假设已经访问过的城市集合是 S,当前所在城市是 v,用 dp[S][v]表示从 v 出发访问剩余的所有城市最后回到起点的路径费用总和的最小值。状态转移方程如下:

$$\text{dp}[V][0]=0 \qquad //V\text{ 是最后一个城市}$$
$$\text{dp}[S][v]=\min(\text{dp}[S\cup\{u\}][u]+\text{dist}(v,u)\mid u\notin S)$$

城市集合 S 如何表示?这就用到状态压缩 DP 的技巧:把路径"压缩"到一个二进制数中。定义:

```
int dp[1 << MAXN][MAXN];
```

MAXN 是城市数量,当 MAXN=15 时,1 << MAXN=2^{15}=32 768,0~32 768 内的每个数的二进制表示就是一个可能的路径,二进制数中的 1 表示选中一个城市,0 表示不选中。例如 S=$000\ 0000\ 0000\ 0101_2$,末尾的 101 表示已访问过城市 2、0。在下面的代码中,"dp[s|1 << u][u]",其中的 s|1 << u,表示在已访问过的城市集合 S 中加入一个新访问的城市 u。

下面是部分示意代码[1]。

```
int dp[1 << MAXN][MAXN];
void solve(){
    memset(dp, INF, sizeof(dp));            //初始化为无穷大
    dp[(1 << n) - 1][0] = 0;                //从最后一个点出发到起点 0,已经没有城市可
                                            //以走,所以到起点 0 的最小路径费用是 0
    for (int s = (1 << n) - 2; s >= 0; s-- ) //O(2^n)
        for(int v = 0; v < n; v++)          //O(n)
            for(int u = 0; u < n; u++)      //O(n)
                if(!(s >> u & 1))
                    dp[s][v] = min(dp[s][v], dp[s|1 << u][u] + dist[v][u]);
    printf("%d\n",dp[0][0]);
}
```

4. 例题 2

这一题是 TSP 的变形。

hdu 3001 "Travelling"

Acmer 先生决定访问 n 座城市。他可以空降到任意城市,然后开始访问,要求访问到所有城市,任何一个城市访问的次数不少于 1 次,不多于 2 次。n 座城市间有 m 条道路,每条道路都有路费。求 Acmer 先生完成旅行需要花费的最小费用。

输入:第 1 行是 n 和 m,$1\leqslant n\leqslant 10$;后面有 m 行,有 3 个整数 a、b、c,表示城市 a 和 b 之间的路费是 c。

输出:最少花费,如果不能完成旅行,则输出−1。

[1] 编程细节参考《挑战程序设计竞赛》(秋叶拓哉)192 页,3.4.1 节。

本题 $n=10$，数据很小，但是由于每个城市可以走两遍，可能的路线就变成了 $(2n)!$，所以不能用暴力法。

本题用状态压缩 DP 求解，算法复杂度是 $O(3^n n^2)$，当 $n=10$ 时正好通过 OJ 测试。

1）路径的表示

在普通 TSP 中，一个城市只有两种情况，即访问和不访问，用 1 和 0 表示。这个题有 3 种情况，也就是不访问、访问 1 次、访问 2 次，所以需要用到三进制。

当 $n=10$ 时有 3^{10} 种组合（路径数量），对每种路径用三进制表示。例如，第 14 种路径，它的三进制是 112_3，表示第 3 个城市走 1 次，第 2 个城市走 1 次，第 1 个城市走 2 次。

在程序中用 tri[i][j]表示第 i 个路径，其第 j 位的值是城市状态，例如 tri[14][3]=1，tri[14][2]=1，tri[14][1]=2。

2）状态和状态转移

定义状态 dp[i][j]，当前所在城市是 i，dp[i][j]表示从 i 出发访问剩余的所有城市最后回到起点的路径费用总和的最小值。

状态转移：dp[j][i]=min(dp[j][i]，dp[k][l]+graph[k][j])；

```
#include <bits/stdc++.h>
const int INF = 0x3f3f3f3f;
using namespace std;
int n,m;
int bit[12] = {0,1,3,9,27,81,243,729,2187,6561,19683,59049};
                                   //三进制每一位的权值,与二进制的 0、1、2、4、8 等对照
int tri[60000][11];
int dp[11][60000];
int graph[11][11];                        //存图
void make_trb(){                          //初始化,求所有可能的路径
    for(int i = 0;i < 59050;++i){         //共 3^10 = 59 050 种状态
        int t = i;
        for(int j = 1; j <= 10; ++j){tri[i][j] = t % 3; t/ = 3;}
    }
}
int comp_dp(){
        int ans = INF;
        memset(dp, INF, sizeof(dp));
        for(int i = 0;i <= n;i++)
            dp[i][bit[i]] = 0;            //bit[i]是第 i 个城市,起点是任意的
        for(int i = 0;i < bit[n + 1];i++){
            int flag = 1;                 //所有的城市都遍历过 1 次以上
            for(int j = 1;j <= n;j++){    //选一个终点
                if(tri[i][j] == 0){       //判断终点位是否为 0
                    flag = 0;             //还没有经过所有点
                    continue;
                }
                if(i == j) continue;
                for(int k = 1; k <= n; k++){
                    int l = i - bit[j];   //i 状态的第 j 位置 0
                    if(tri[i][k] == 0) continue;
```

```
                dp[j][i] = min(dp[j][i],dp[k][l] + graph[k][j]);
            }
        }
        if(flag)                                //找最小费用
            for(int j = 1; j <= n; j++)
                ans = min(ans,dp[j][i]);
    }
    return ans;
}
int main(){
    make_trb();
    while(cin >> n >> m){
        memset(graph,INF,sizeof(graph));
        while(m--){
            int a,b,c;
            cin >> a >> b >> c;
            if(c < graph[a][b])  graph[a][b] = graph[b][a] = c;
        }
        int ans = comp_dp();
        if(ans == INF) cout <<"-1"<< endl;
        else           cout << ans << endl;
    }
    return 0;
}
```

【习题】

hdu 1074 “Doing Homework”，入门题。

hdu 2167 “Pebbles”。

hdu 3182 “Hamburger Magi”。

hdu 4539 “排兵布阵”。

poj 1185 “炮兵阵地”，经典题。

poj 2411 “Mondriaan's Dream”，铺砖问题。

hdu 3681 “Prison Break”，TSP＋二分，难题。

7.7 小　　结

本章介绍了常见的 DP 算法。读者已经看到，DP 题目不仅涉及大量知识点，而且思维灵活，不容易掌握。DP 题目作为竞赛的必考题型，参赛者需要花大量时间练习，掌握其中的诀窍。

另外还有很多可用 DP 的算法在本章没有涉及，例如用 DP 解决以概率为最优解的问题，具体内容见本书 8.4 节；还有 AC 自动机＋DP、后缀自动机＋DP 等。

第8章 数　　学

✍高精度计算

✍数论

✍组合数学

✍概率和数学期望

✍公平组合游戏

数学题在算法竞赛中经常出现。数学题的知识点相当广，有些容易理解，有些比较难。在竞赛中经常把数学模型和其他算法结合起来，出综合性的题目，所以本书把数学题相关内容放在比较靠后的章节。

常见的数学方面的题目包括数论、组合数学、概率和数学期望、组合游戏等大类，这里先列出常见的知识点，本章将讲解其中部分内容。

（1）数论。

整除性问题：整除、最大公约数、最大公倍数；欧几里得算法、扩展欧几里得算法。

素数问题：素数判定、区间素数统计。

同余问题：模运算、同余方程、快速幂、中国剩余定理、逆元、整数分解、同余定理。

不定方程。

乘性函数：欧拉函数、伪随机数、莫比乌斯反演。

（2）组合数学。

排列组合：计数原理、特殊排列、排列生成、组合生成。

母函数：普通型、指数型。

递推关系：Fibonacci 数列、Stirling 数、Catalan 数。

容斥原理、鸽巢原理。

群：Polya 定理。

线性规划：单纯形法。

（3）矩阵、线性代数、高精度计算、概率和数学期望、组合游戏、傅里叶变换。

8.1　高精度计算

视频讲解

高精度计算，是指参与运算的数大大超出了标准数据类型所能表示的范围的运算，例如两个 1000 位数相乘。这类题目在算法竞赛中的出现很频繁。

在 C 或者 C++ 中，最大的数据类型只有 64 位，如果需要处理更大的数，只能用数组来模

拟,把大数的每一位存储在数组中,然后按位处理进位、借位问题,相当麻烦。

但是用 Java 处理高精度非常简单,可以直接计算。在 Java 中有两个类——BigInteger 和 BigDecimal,分别表示大整数类和大浮点数类,两个类的对象能处理的数理论上能够表示无限大,只要计算机内存足够大。这两个类都在 java.math.* 包中。

例如 hdu 1042 题,输入整数 $N(0 \leqslant N \leqslant 10\ 000)$,输出 $N!$。当 $N=10\ 000$ 时,$N!$ 是一个超级大的数字。读者可以尝试用 C++实现①。用 Java 可以直接算,下面是代码。

hdu 1042 题的 Java 代码

```java
import java.math.BigInteger;
import java.util.*;
public class Main{
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        while(input.hasNext()) {
            int n = input.nextInt();
            BigInteger ans = BigInteger.ONE;
            for(int i = 1; i <= n; i++)
                ans = ans.multiply(BigInteger.valueOf(i));
            System.out.println(ans);
        }
    }
}
```

Java 虽然能处理大数,但是对于规模过大的问题用 Java 也不能做。例如 hdu 1061 题,$n=10^9$,求 n^n。此时需要一些特殊的算法,例如"快速幂",见下一节内容。

【习题】

请读者自己找资料熟悉 Java 的高精度运算,并通过以下习题掌握用法。

hdu 1047,求和。

hdu 1063,实数的高精度幂。

hdu 1316,大数比较。

hdu 5666,大数除法。

hdu 5686,大数递推。

8.2 数 论

数论是研究整数性质的数学分支。初等数论的主要内容有整除问题、素数、不定方程、同余问题、乘性函数等。本节介绍竞赛中常用的一些初等数论知识。

① 用"万进制"可以求解 $n!$,请读者搜索网上资料。

8.2.1 模运算

模运算是大数运算中的常用操作。如果一个数太大，无法直接输出，或者不需要直接输出，可以把它取模后缩小数值再输出。

定义取模运算为 a 除以 m 的余数①，记为：

$$a \bmod m = a \ \% \ m$$

取模的结果满足 $0 \leqslant a \bmod m \leqslant m-1$，题目用给定的 m 限制计算结果的范围。例如 $m=10$，就是取计算结果的个位数，参考 hdu 1061 题，求 n^n，$n \leqslant 10^9$，输出结果的个位数。

取模操作满足以下性质。

加：$(a+b) \bmod m = ((a \bmod m)+(b \bmod m)) \bmod m$

减：$(a-b) \bmod m = ((a \bmod m)-(b \bmod m)) \bmod m$

乘：$(a \times b) \bmod m = ((a \bmod m) \times (b \bmod m)) \bmod m$

然而，对除法取模进行下面的类似操作是错误的：

$$(a/b) \bmod m = ((a \bmod m)/(b \bmod m)) \bmod m$$

例如，$(100/50) \bmod 20 = 2$，$(100 \bmod 20)/(50 \bmod 20) \bmod 20 = 0$，两者不相等。

除法的取模需要用到逆元，将在“同余与逆元”这一节中介绍。

8.2.2 快速幂

1. 快速幂概念

快速幂以及扩展的矩阵快速幂，由于应用场景比较常见，也是竞赛中常见的题型。

幂运算 a^n 即 n 个 a 相乘。快速幂就是高效地算出 a^n。当 n 很大时，例如 $n=10^9$，计算 a^n 这样大的数 Java 也不能处理，一是数字太大，二是计算时间很长。下面先考虑如何缩短计算时间，如果用暴力的方法直接算 a^n，即逐个做乘法，复杂度是 $O(n)$，即使能算出来，也会超时。

读者很容易想到快速幂的办法：先算 a^2，然后继续算平方 $(a^2)^2$，一直算到 n 次幂。这是分治法的思想，复杂度为 $O(\log_2 n)$。下面是代码，请读者自己理解：

```
int fastPow(int a, int n){
    if(n == 1)   return a;
    int temp = fastPow(a, n/2);          //分治
    if(n % 2 == 1)                        //奇数个 a,此处也可以写为 if(n &1)
        return temp * temp * a;
    else                                  //偶数个 a
        return temp * temp;
}
```

程序中的递归，层数只有 $\log_2 n$，不用担心溢出的问题。

上面的程序非常好，不过还有一种更好的方法，是用位运算做快速幂，时间复杂度也是 $O(\log_2 n)$。下面以 a^{11} 为例说明快速幂的原理。

① 注意，此时要求 a 和 m 的正负号一致，都为正数或都为负数。如果正负不同，取模和求余的结果是不同的。

先把 a^{11} 分解成 a^8、a^2、a^1 的乘积，即 $a^{11}=a^{8+2+1}=a^8\times a^2\times a^1$。

如何求 a^8、a^2、a^1 的值，需要分别计算吗？并不需要。用户可以容易地发现，$a^1\times a^1=a^2$，$a^2\times a^2=a^4$，$a^4\times a^4=a^8$，等等，都是 2 的倍数，产生的 a^i 都是倍乘关系，逐级递推就可以了。在下面的程序中，这个功能用“base *=base;”实现。

那么如何把 n 分解成 11=8+2+1 这样的倍乘关系？用二进制就能理解了。把 n 转为二进制数，二进制数中每一位的权值都是低一位的两倍，对应的 a^i 是倍乘的关系，例如 $n=11_{10}=1011_2=2^3+2^1+2^0=8+2+1$，所以只需要把 n 按二进制处理就可以了。

另外还有一个需要处理的问题：如何跳过那些不需要的？例如求 a^{11}，因为 11=8+2+1，需要跳过 a^4。这里做个判断即可，1011 中的 0 就是需要跳过的。这个判断，利用二进制的位运算很容易实现：

(1) n & 1，取 n 的最后一位，并且判断这一位是否需要跳过。

(2) n >>=1，把 n 右移一位，目的是把刚处理过的 n 的最后一位去掉。

```
int fastPow(int a, int n){
    int base = a;                      //不定义 base,直接用 a 进行计算也行
    int res = 1;                       //用 res 返回结果
    while(n) {
        if(n & 1)                      //如果 n 的最后一位是 1,表示这个地方需要乘
            res * = base;
        base * = base;                 //推算乘积,a² --> a⁴ --> a⁸ --> a¹⁶ …
        n >>= 1;                       //n 右移一位,把刚处理过的 n 的最后一位去掉
    }
    return res;
}
```

对照上面的程序，执行步骤如表 8.1 所示。

表 8.1　执行步骤

	n	res(res *=base)	base(base *=base)
第 1 轮	1011	a^1	a^2
第 2 轮	101	$a^1\times a^2$	a^4
第 3 轮	10	是 0，res 不变	a^8
第 4 轮	1	$a^1\times a^2\times a^8$	a^{16}
结束	0		

2. 快速幂取模

由于幂运算的结果非常大，常常会超过变量类型的最大值，甚至超过内存所能存放的最大数，所以涉及快速幂的题目，通常都要做取模操作，缩小结果。

根据模运算的性质，在快速幂中做取模操作，对 a^n 取模，和先对 a 取模再做幂运算的结果是一样的，即：

$$a^n \bmod m=(a \bmod m)^n \bmod m$$

下面修改位运算 fastPow()函数，加上取模操作。以 hdu 2817 题为例，取模操作如下：

```
const int mod = 200907;
…
```

```
    if(n & 1)
        res = (base * res) % mod;
    base = (base * base) % mod;
    ...
```

对于分治法 fastPow()函数的取模操作,请读者自己做类似的修改。

3. 矩阵快速幂

给定一个 $m \times m$ 的矩阵 $\boldsymbol{A}$,求它的 n 次幂 $\boldsymbol{A}^n$,这也是常见的计算。同样有矩阵快速幂的算法,原理是把矩阵当作变量来操作,程序和上面的很相似。

首先需要定义矩阵的结构体,并且定义矩阵相乘的操作。注意矩阵相乘也需要取模。

```
const int MAXN = 2;                       //根据题目要求定义矩阵的阶,本例中是 2
const int MOD = 1e4;                      //根据题目要求定义模
struct Matrix{                            //定义矩阵
    int m[MAXN][MAXN];
    Matrix() {
        memset(m, 0, sizeof(m));
    }
};
Matrix Multi(Matrix a, Matrix b) {        //矩阵的乘法
    Matrix res;
    for(int i = 0; i < MAXN; i++)
        for(int j = 0; j < MAXN; j++)
            for(int k = 0; k < MAXN; k++)
                res.m[i][j] = (res.m[i][j] + a.m[i][k] * b.m[k][j]) % MOD;
    return res;
}
```

下面是矩阵快速幂的程序代码,和前面单变量的快速幂的代码非常相似。

```
Matrix fastm(Matrix a, int n){
    Matrix res;
    for(int i = 0; i < MAXN; i++)
                //初始化为单位矩阵,相当于前面程序中的"int res = 1;"
      res.m[i][i] = 1;
    while(n) {
        if(n&1)
            res = Multi(res, a);
        a = Multi(a, a);
        n >>= 1;
    }
    return res;
}
```

矩阵快速幂的复杂度:上面求 $\boldsymbol{A}^n$,$\boldsymbol{A}$ 是 $m \times m$ 的方阵,其中矩阵乘法的复杂度是 $O(m^3)$,快速幂的复杂度是 $O(\log_2 n)$,合起来是 $O(m^3 \log_2 n)$。

应用矩阵快速幂的难点在于如何把递推关系转换为矩阵。

【习题】

hdu 2817。

hdu 1061，求 n^n 的末尾数字，$n \leqslant 10^9$。

hdu 5392，快速幂取模、LCM。

poj 3070、hdu 3117。矩阵快速幂的经典题目，算 Fibonacci 数列。求第 10^9 个 Fibonacci 数，因为直接递推无法完成，所以先用矩阵表示 Fibonacci 数列的递推关系，然后把问题转换为求这个矩阵的 10^9 幂。

hdu 6030，把递推关系转换为矩阵。

hdu 5895，有难度的矩阵快速幂。

hdu 5564，数位 DP、矩阵快速幂。

hdu 2243，AC 自动机、矩阵快速幂。

8.2.3 GCD、LCM

最大公约数 GCD 和最小公倍数 LCM 是竞赛中常见的知识点，虽然这两个知识点很容易理解，但往往会与其他知识点结合起来出综合题，并不容易。

1. 最大公约数 GCD

整数 a 和 b 的最大公约数记为 $\gcd(a,b)$。在编程时有两种做法。

(1) 经典的欧几里得算法，用辗转相除法求最大公约数，模板如下：

```
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}
```

时间复杂度差不多是 $O(\log_2 n)$ 的①，非常快。

(2) 或者直接用 C++的内置函数求 GCD：

```
std::__gcd(a, b)
```

2. 最小公倍数 LCM

整数 a 和 b 的最小公倍数记为 $\mathrm{lcm}(a,b)$，模板如下：

```
int lcm(int a, int b) {
    return a/gcd(a, b) * b;
}
```

8.2.4 扩展欧几里得算法与二元一次方程的整数解

读者可能还记得中学接触过的一个问题：给出整数 a、b、n，问方程 $ax+by=n$ 什么时

① 严格的复杂度分析参考《初等数论及其应用》第 6 版，Kenneth H. Rosen 著，夏鸿刚译，机械出版社，3.4 节的欧几里得算法。

候有整数解？如何求所有的整数解？

有解的充分必要条件是 gcd(a,b)可以整除 n。简单解释如下：

令 $a=\gcd(a,b)a'$、$b=\gcd(a,b)b'$，有 $ax+by=\gcd(a,b)(a'x+b'y)=n$；如果 x、y、a'、b'都是整数，那么 n 必须是 gcd(a,b)的倍数才有整数解。

例如 $4x+6y=8$、$2x+3y=4$ 有整数解，$4x+6y=7$ 则没有整数解。

如果确定有解，一种解题方法是先找到一个解(x_0,y_0)，那么通解公式如下：

$$x = x_0 + bt$$
$$y = y_0 - at,\quad t\text{ 是任意整数}$$

所以，问题转化为如何求(x_0,y_0)。利用扩展欧几里得算法可以求出这个特解。

1. 扩展欧几里得算法

当方程符合 $ax+by=\gcd(a,b)$时，可以用扩展欧几里得算法求(x_0,y_0)。程序如下[①]：

```
void extend_gcd(int a, int b, int &x, int &y){      //返回 x,y,即一个特解(x0,y0)
    if(b == 0) {
        x = 1, y = 0;
        return;
    }
    extend_gcd(b, a % b, x, y);
    int tmp = x;
    x = y;
    y = tmp - (a/b) * y;
}
```

有时候为了简化描述，在 $ax+by=\gcd(a,b)$两边除以 gcd(a,b)，得到 $cx+dy=1$，其中 $c=a/\gcd(c,b)$，$d=b/\gcd(a,b)$。很明显，c、d 是互质的。$cx+dy=1$ 的通解如下：

$$x = x_0 + dt$$
$$y = y_0 - ct,\quad t\text{ 是任意整数}$$

2. 求任意方程 $ax+by=n$ 的一个整数解

用扩展欧几里得算法求解 $ax+by=\gcd(a,b)$后，利用它可以进一步解任意方程 $ax+by=n$，得到一个整数解。其步骤如下：

(1) 判断方程 $ax+by=n$ 是否有整数解，有解的条件是 gcd(a,b)可以整除 n；

(2) 用扩展欧几里得算法求 $ax+by=\gcd(a,b)$的一个解(x_0,y_0)；

(3) 在 $ax_0+by_0=\gcd(a,b)$两边同时乘以 $n/\gcd(a,b)$，得：

$$ax_0n/\gcd(a,b) + by_0n/\gcd(a,b) = n$$

(4) 对照 $ax+by=n$，得到它的一个解(x_0',y_0')是：

$$x_0' = x_0n/\gcd(a,b)$$
$$y_0' = y_0n/\gcd(a,b)$$

3. 应用场合

扩展欧几里得算法是一个很有用的工具，在竞赛题目中常用于以下场合：

① 程序的执行过程参考《算法导论》，Thomas H. Cormen 等著，机械工业出版社，31.2 节。

(1) 求解不定方程；

(2) 求解模的逆元；

(3) 求解同余方程。

虽然用扩展欧几里得算法可以算 $ax+by=\gcd(a,b)$ 的通解，不过一般没有这个需求，而是用于求某些特殊的解，例如求解逆元，逆元是除法取模操作常用的工具。

【习题】

poj 1061，扩展欧几里得。

hdu 1019，LCM。

hdu 1576，扩展欧几里得。

hdu 2504，GCD，水题。

hdu 2588，GCD，欧拉函数。

hdu 5223，GCD，贪心。

hdu 5584，LCM。

hdu 5656，GCD，DP。

hdu 5902，GCD，暴力。

8.2.5　同余与逆元

同余在数论中非常有用，它用类似处理等式的方式来处理整除关系，非常简便。

1. 同余

两个整数 a、b 和一个正整数 m，如果 a 除以 m 所得的余数和 b 除以 m 所得的余数相等，即 $a \bmod m=b \bmod m$，称 a 和 b 对于 m 同余[①]，m 称为同余的模。同余的概念也可以这样理解：$m|(a-b)$，即 $a-b$ 是 m 的整数倍。例如 $6|(23-5)$，23 和 5 对模 6 同余。

同余的符号记为 $a\equiv b(\mathrm{mod}\ m)$。

2. 一元线性同余方程

$ax\equiv b(\mathrm{mod}\ m)$，即 ax 除以 m，b 除以 m，两者余数相同，这里 a、b、m 都是整数，求解 x 的值。

方程也可以这样理解：$ax-b$ 是 m 的整数倍。设 y 是倍数，那么 $ax-b=my$，移项得到 $ax-my=b$。因为 y 可以是负数，改写为 $ax+my=b$，这就是在扩展欧几里得算法中提到的二元一次不定方程。

当且仅当 $\gcd(a,m)$ 能整除 b 时有整数解。例如 $15x+6y=9$，有整数解 $x=1$，$y=-1$。

当 $\gcd(a,m)=b$ 时，可以直接用扩展欧几里得算法求解 $ax+my=b$。

如果不满足 $\gcd(a,m)=b$，还能用扩展欧几里得算法求解 $ax+my=b$ 吗？答案是肯定的，但是需要结合下面的逆元。

3. 逆元

给出 a 和 m，求解方程 $ax\equiv 1(\mathrm{mod}\ m)$，即 ax 除以 m 余数是 1。

---

① 《初等数论及其应用》第 6 版，Kenneth H. Rosen 著，夏鸿刚译，机械工业出版社，第 4 章“同余”。

根据前面的讨论，有解的条件是 $\gcd(a,m)=1$，即 a、m 互素。该问题等价于求解 $ax+my=1$，可以用上一节的扩展欧几里得算法求解。例如 $8x\equiv1(\text{mod }31)$，等价于求解 $8x+31y=1$，用扩展欧几里得算法求得一个特解是 $x=4,y=-1$。

方程 $ax\equiv1(\text{mod }m)$的一个解 x，称 x 为 a 模 m 的逆。注意，这样的 x 有很多，把它们都称为逆。

求逆元的代码如下：

```
int mod_inverse(int a, int m){
    int x, y;
    extend_gcd(a, m, x, y);
    return(m + x % m) % m;              //x 可能是负数,需要处理
}
```

另外，在某些情况下也可以用费马小定理求逆元。

4. 逆元与除法取模

逆元的一个重要应用是求除法的模。在后面讲 Catalan 数的时候有这样一个需求：求 $(a/b)\text{mod }m$，即 a 除以 b，然后对 m 取模。由于这里 a 和 b 都是很大的数，做除法后再取模会损失精度。下面的方法可以避开除法计算。

设 b 的逆元是 k，有：

$$\left(\frac{a}{b}\right)\text{mod }m=\left(\left(\frac{a}{b}\right)\text{mod }m\right)((bk)\text{mod }m)=\left(\frac{a}{b}bk\right)\text{mod }m=(ak)\text{mod }m$$

上述推导过程把除法的模运算转换成了乘法模运算：$(a/b)\text{mod }m=(ak)\text{mod }m$

5. 逆元与求解二元一次方程 $ax+my=b$

如果得到了 a 的逆，可以来求解形如 $ax\equiv b(\text{mod }m)$的任何同余方程。方法如下：令 a'是 a 模 m 的逆，则 $a'a\equiv1(\text{mod }m)$；在 $ax\equiv b(\text{mod }m)$的两边同时乘以 a'，得到 $a'ax\equiv a'b(\text{mod }m)$，所以 $x\equiv a'b(\text{mod }m)$。

例如求 $8x\equiv24(\text{mod }31)$的解。先求 8 模 31 的逆，是 4；然后在 $8x\equiv24(\text{mod }31)$的两边乘以 4，得到 $8\times4x\equiv4\times24(\text{mod }31)$，所以 $x\equiv96(\text{mod }31)$。

前面讲解扩展欧几里得算法时曾求解了二元一次方程，这里再给出利用逆元的另一种方法，如表 8.2 所示。读者对照两种方法，可以加深对逆元的理解。

表 8.2　利用逆元的求解方法

步骤	求解方程 $ax+my=b$ 同余方程是 $ax\equiv b(\text{mod }m)$	例：求解 $8x+31y=24$ 同余方程是 $8x\equiv24(\text{mod }31)$ $a=8,b=24,m=31$
1	有解的条件：$\gcd(a,m)$能整除 b	$\gcd(8,31)=1$ 能整除 24
2	求 $ax\equiv1(\text{mod }m)$的逆元 a'，等价于用扩展欧几里得算法求解 $ax+my=1$	$8x+31y=1$ 的一个解是 $x=4,y=-1$ 即一个逆元是 $a'=4$
3	一个特解是 $x=a'b$	$x=a'b=4\times24=96$
4	代入方程 $ax+my=b$，求解 y	代入 $8x+31y=24$，得到 $y=-24$

【习题】

hdu 5976 “Detachment”，乘法逆元。

8.2.6　素数

1. 用试除法判断素数

问题：输入一个很大的数 n，判断它是不是素数。

素数定义：一个数 n，如果不能被 $[2,n-1]$ 内的所有数整除，n 就是素数。当然，并不需要把 $[2,n-1]$ 内的数都试一遍，这个范围可以缩小到 $[2,\sqrt{n}]$。

给定 n，如果它不能整除 $[2,\sqrt{n}]$ 内的所有数，它就是素数。证明如下：

设 $n=a\times b$，有 $\min(a,b)\leqslant\sqrt{n}$，令 $a\leqslant b$。只要检查 $[2,\sqrt{n}]$ 内的数，如果 n 不是素数，就能找到一个 a。如果不存在这个 a，那么 $(\sqrt{n},n-1]$ 内也不存在 b。

以上判断的范围可以再缩小一点：$[2,\sqrt{n}]$ 内所有的**素数**。其原理很简单，读者在学过下文提到的埃式筛法之后更容易理解。

用试除法判断素数，复杂度是 $O(\sqrt{n})$，对于 $n\leqslant 10^{12}$ 的数是没有问题的。

下面是试除法判断素数的代码。

判断素数

```
bool is_prime(int n){
    if(n<=1)   return false;              //1 不是素数
    for(int i=2; i*i<=n; i++)             //比这样写更好: for(int i=2;i<=sqrt(n);i++)
        if(n % i == 0)  return false;     //能整除,不是素数
    return true;
}
```

2. 巨大素数的判断

如果 n 非常大，例如 poj 1811 题，$1\leqslant n<2^{54}$，判断 n 是不是素数。如果用试除法，$\sqrt{n}=2^{27}\approx 10^8$，复杂度仍然太高。此时需要用到特殊而复杂的方法，如果读者有兴趣，可以自己查资料①。

3. 用埃式筛法求素数的数量

一个与素数相关的问题是求 $[2,n]$ 内所有的素数。如果用上面的试除法，一个个单独进行判断，太慢了。

埃式筛法是一种古老而简单的方法，可以快速找到 $[2,n]$ 内所有的素数。对于初始队列 $\{2,3,4,5,6,7,8,9,10,11,12,13,\cdots,n\}$，操作步骤如下：

(1) 输出最小的素数 2，然后筛掉 2 的倍数，剩下 $\{3,5,7,9,11,13,\cdots\}$；

① 《ACM/ICPC 算法训练教程》，余立功，清华大学出版社，127 页。

(2) 输出最小的素数 3,然后筛掉 3 的倍数,剩下{5,7,11,13,…};

(3) 输出最小的素数 5,然后筛掉 5 的倍数,剩下{7,11,13,…}。

继续以上步骤,直到队列为空。

下面是程序,其中 visit[i]记录数 i 的状态,如果 visit[i]=true,表示它被筛掉了,不是素数。用 prime[]存放素数,例如 prime[0]是第 1 个素数 2。

```
const int MAXN = 1e7;                        //定义空间大小,1e7 约 10MB
int prime[MAXN + 1];                         //存放素数,它记录 visit[i] = false 的项
bool visit[MAXN + 1];                        //true 表示被筛掉,不是素数
int E_sieve(int n)  {                        //埃式筛法,计算[2, n]内的素数
   int k = 0;                                //统计素数的个数
   for(int i = 0; i <= n; i++)  visit[i] = false;    //初始化
   for(int i = 2; i <= n; i++) {             //从第 1 个素数 2 开始.可优化(1)
      if(!visit[i]) {
         prime[k++] = i;                     //i 是素数,存储到 prime[]中
         for(int j = 2 * i; j <= n; j += i)  //i 的倍数都不是素数。可优化(2)
            visit[j] = true;                 //标记为非素数,筛掉
      }
   }
   return k;                                 //返回素数的个数
}
```

计算复杂度:2 的倍数被筛掉,计算 $n/2$ 次;3 的倍数被筛掉,计算 $n/3$ 次;5 的倍数被筛掉,计算 $n/5$ 次,依此类推。总次数是 $O(n/2+n/3+n/5+\cdots)$,这里直接给出结果,即 $O(n\log\log_2 n)$。

空间复杂度:程序用到了 bool visit[MAXN+1]数组,当 MAXN=10^7 时约 10MB。一般题目会限制空间为 65MB,所以 n 不能再大了。

上述代码有两处可以优化:

(1) 用来做筛除的数为 2、3、5 等,最多到$\sqrt{n}$就可以了。例如求 $n=100$ 以内的素数,用 2、3、5、7 筛就足够了。其原理和试除法一样:非素数 k 必定可以被一个小于等于$\sqrt{k}$的素数整除。

(2) for(int j=2 * i; j<=n; j+=i)中的 $j=2*i$ 优化为 $j=i*i$。例如 $i=5$ 时,2 * 5、3 * 5、4 * 5 已经在前面 $i=2,3,4$ 的时候筛过了。

优化后的代码如下:

```
int E_sieve(int n) {
   for(int i = 0; i <= n; i++)  visit[i] = false;
   for(int i = 2; i * i <= n; i++)           //筛掉非素数
      if(!visit[i])
         for(int j = i * i; j <= n; j += i)
            visit[j] = true;                 //标记为非素数
      //下面记录素数
   int  k = 0;                               //统计素数的个数
   for(int i = 2; i <= n; i++)
      if(!visit[i])
```

```
        prime[k++] = i;                    //存储素数
    return k;
}
```

埃式筛法虽然还不错，但其实做了一些无用功，某个数会被筛几次，比如 12，被 2 和 3 筛了两次。另一种欧拉筛选法，时间复杂度仅为 $O(n)$，如果读者有兴趣可以查资料。不过，埃式筛法可以近似看成 $O(n)$ 的，一般也够用了。

4. 埃式筛法应用于大区间素数

用埃式筛法求$[2,n]$内的素数，只能解决规模 $n\leqslant10^7$ 的问题。如果 n 更大，在某些情况下可以用埃式筛法来处理，这就是大区间素数的计算。

如果把$[2,n]$看成一个区间，那么可以把埃式筛法扩展到求区间$[a,b]$的素数，$a<b\leqslant10^{12}$，$b-a\leqslant10^6$。

前文提到，用试除法判断 n 是不是素数，原理为：如果它不能整除 $2\sim\sqrt{n}$ 内所有的素数，它就是素数。根据埃式筛法很容易理解这个原理：$2\sim\sqrt{n}$ 内的非素数 b 肯定对应一个比它小的素数 a。在用试除法的时候，如果 n 能整除 a，已经证明了 n 不是素数，b 就不用再试了。

这个原理可以用来理解大区间求素数问题。先用埃式筛法求$[2,\sqrt{b}]$内的素数，然后用这些素数来筛$[a,b]$区间的素数即可。

(1) 计算复杂度：$O(\sqrt{b}\log\log\sqrt{b})+O((b-a)\sqrt{b-a})$；

(2) 空间复杂度：需要定义两个数组，一个用于处理$[2,\sqrt{b}]$内的素数，另一个用于处理$[a,b]$内的素数，空间复杂度是 $O(\sqrt{b})+O(b-a)$。

习题：poj 2689，求$[L,R]$内的素数，$1\leqslant L<R\leqslant2\,147\,483\,647$)，$R-L\leqslant10^6$。

5. 更大的素数

上面埃式筛法的限制条件是 $n\leqslant10^7$。如果要统计更大范围内的素数个数，例如 $n=10^{11}$时有 40 多亿个素数[①]，此时需要用到更复杂的数学方法。如果读者有兴趣可以研究 hdu 5901 Count primes 一题，求 $1\leqslant n\leqslant10^{11}$范围内的素数个数。

【习题】

hdu 1262，寻找素数对。

hdu 2710，筛法求素数。

hdu 3792，素数打表。

hdu 3826，分解质因子。

hdu 6069，区间素数。

① $[2,n]$内素数的数量：https://en.wikipedia.org/wiki/Prime-counting_function（永久网址：perma.cc/MSN6-F4AM）。

8.3 组合数学

人们在生活中经常会遇到排列组合问题。简单的，例如在 5 个礼物中选两个，问有多少种选取方法？复杂一点的，例如一串手环，用不同颜色的珠子串成，问有多少种不同的排列方法？

组合数学就是研究一个集合内满足一定规则的排列问题。这类问题如下：

(1) 存在问题，即判断这些排列是否存在；

(2) 计数问题，计算出有多少种排列，并构造出来；

(3) 优化问题，如果有最优解，给出最优解。

组合数学涉及的内容很多，包括①：

(1) 基本计数规则，例如乘法规则、加法规则、生成排列组合、多项式系数、鸽巢(抽屉)原理等。

(2) 计数问题，例如母函数(普通型、指数型、概率型等)、二项式定理、递推关系、容斥定理、Pólya 定理等。

(3) 存在问题，例如编码、组合设计、图论中的存在问题等。

(4) 组合优化，例如匹配和覆盖、图和网络的优化问题。

本书的内容涉及前两部分，即计数规则和计数问题。

8.3.1 鸽巢原理

鸽巢原理(Pigeonhole Principle)，或称抽屉原理(Drawer Principle)，内容非常简单：把 $n+1$ 个物体放进 n 个盒子，至少有一个盒子包含两个或更多的物体。

鸽巢原理是很基本的组合原理，但是可以解决许多有趣的问题，得到一些有趣的结论。例如：在 1500 人中，至少有 5 人生日相同；n 个人互相握手，一定有两个人握手的次数相同。

hdu 1205 "吃糖果"

Gardon 有 K 种糖果，每种数量已知，Gardon 不喜欢连续两次吃同样的糖果，问有没有可行的吃糖方案。

该题是非常典型的鸽巢原理问题，可以用"隔板法"求解。找出最多的一种糖果，把它的数量 N 看成 N 个隔板，隔成 N 个空间(把每个隔板的右边看成一个空间)；其他所有糖果的数量为 S。

(1) 如果 $S<N-1$，把 S 个糖果放到隔板之间，这 N 个隔板不够放，必然至少有两个隔

① 参考《应用组合数学》，Fred S. Roberts、Barry Tesman 著，冯速译，机械工业出版社。这本书几乎包罗了所有的组合数学知识。这类书的特点是过于详细，读者不可能通读所有内容，也不太容易提炼出一些知识点的算法思想。建议读者边做竞赛题边查阅有关知识，这样能很快地把知识与应用结合起来。

板之间没有糖果，由于这两个隔板是同一种糖果，所以无解。

(2) 当 $S \geqslant N-1$ 时，肯定有解。其中一个解是把 S 个糖果排成一个长队，注意同种类的糖果是挨在一起的，然后每次取 N 个糖果，按顺序一个一个地放进 N 个空间。由于隔板的数量比每一种糖果的数量都多，所以不可能有两个同样的糖果被放进一个空间里。把 S 个糖果放完，就是一个解，一些隔板里面可能放几种糖果。

鸽巢原理是 Ramsey 定理的一个特例。读者可以通过这两题来了解 Ramsey 定理，即 hdu 5917/6152，它们是 2016、2017 年的比赛题。

【习题】

poj 2356/3370。

hdu 1808/3183/5776。

8.3.2　杨辉三角和二项式系数

读者一定非常熟悉排列和组合公式。

排列：$A_n^k = \dfrac{n!}{(n-k)!}$

组合：$C_n^k = \dbinom{n}{k} = \dfrac{A_n^k}{k!} = \dfrac{n!}{k!(n-k)!}$

这里把组合数 C_n^k 用符号 $\dbinom{n}{k}$ 表示，称为二项式系数(Binomial Coefficient)。

杨辉三角(国外称帕斯卡三角)是二项式系数 $\dbinom{n}{r}$ 的典型应用。

杨辉三角是排列成如下三角形的数字：

$$
\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1
\end{array}
$$

每一行从上一行推导而来。如果编程求杨辉三角第 n 行的数字，可以模拟这个推导过程，逐级递推，复杂度是 $O(n^2)$。不过，若改用数学公式计算，则可以直接得到结果，比用递推快多了，这个公式就是 $(1+x)^n$。

观察 $(1+x)^n$ 的展开：

$$
\begin{aligned}
(1+x)^0 &= 1 \\
(1+x)^1 &= 1+x \\
(1+x)^2 &= 1+2x+x^2 \\
(1+x)^3 &= 1+3x+3x^2+x^3
\end{aligned}
$$

每一行展开的系数刚好对应杨辉三角每一行的数字。也就是说，杨辉三角可以用

$(1+x)^n$ 来定义和计算。

那么如何计算 $(1+x)^n$？真的需要展开算系数吗？并不需要，二项式系数 $\binom{n}{k}=\frac{n!}{k!(n-k)!}$ 就是 $(1+x)^n$ 展开后的系数。它们的关系可以这样理解：$(1+x)^n$ 的第 k 项，实际上就是从 n 个 x 中选出 k 个，这就是组合数 $\binom{n}{k}$ 的定义。所以：

$$(1+x)^n=\sum_{k=1}^{n}\binom{n}{k}x^k$$

这个公式称为二项式定理。

有了这个公式，在求杨辉三角第 n 行的数字时就可以用公式直接计算了，复杂度为 $O(1)$。不过，该公式中有 $n!$，如果直接计算 $n!$，由于太大，有可能溢出。例如 hdu 2032 题，$n=30$，30！超过了 long long 的范围。此时可以利用 $\binom{n}{k-1}$ 和 $\binom{n}{k}$ 的递推关系 $\frac{\binom{n}{k}}{\binom{n}{k-1}}=\frac{n-k+1}{k}$ 逐个推导，避免计算阶乘。

8.3.3 容斥原理

容斥原理(Inclusion-Exclusion Principle)是常见的思维方法。在计数时，有时情况比较多，相互有重叠。为了使重叠部分不被重复计算，可以这样处理：先不考虑重叠的情况，把所有对象的数目计算出来，然后减去重复计算的数目。这种计数方法称为容斥原理。

例如一根长为 60m 的绳子，每隔 3m 做一个记号，每隔 4m 也做一个记号，然后把有记号的地方剪断，问绳子共被剪成了多少段？容斥原理的解题思路是：3 的倍数有 20 个，不算绳子两头，有 20－1＝19 个记号；4 的倍数有 15 个；既是 3 的倍数又是 4 的倍数的，有 60÷(3×4)＝5 个。所以记号的总数量是(20－1)＋(15－1)－(5－1)＝29，绳子被剪成 29 段。

【习题】

hdu 2841/4135/4497/5155。

8.3.4 Fibonacci 数列

Fibonacci 数列是一个很常见的递推数列，在小学奥数中被称为“兔子数列”。Fibonacci 数列也是一个被“神话”的数列，人们常常提到的“黄金分割”就蕴含在 Fibonacci 数列中。

1. Fibonacci 数列的递推公式

$$f(1)=f(2)=1$$
$$f(n)=f(n-1)+f(n-2)$$

从第 3 项开始，每一项都等于前 2 项之和，前一部分数是 1,1,2,3,5,8,13…

当 n 趋于无穷大时，相邻两个数的比值 $f(n)/f(n-1) \to 0.618\,033\,988\,7\cdots$ 这就是有名的黄金分割数[①]。

2. 计算 Fibonacci 数列

这里有两个问题：①如何更快地计算第 n 个 Fibonacci 数；②Fibonacci 数增长太快了，需要处理大数。

那么如何计算 Fibonacci 数列？如果只计算到第 10^6 个数，用上面的递推公式就可以，复杂度是 $O(n)$。如果更大，例如算第 1 亿个数，这样就比较慢。对于这么大的 Fibonacci 数，需要用一种巧妙的方法：把递推关系转换成矩阵，并用前面讲过的矩阵快速幂进行处理。请读者自己做 poj 3070 题和 hdu 3117 题，求第 1 亿个 Fibonacci 数，这是一个**必做题**。

另外，Fibonacci 数的值增长非常快，近似于 $O(2^n)$，例如第 40 个数是 102 334 155，已经非常大，所以常常需要处理大数，或者做取模操作。由于这些知识在前面讲过，这里不再赘述。

3. 应用模型

Fibonacci 数列看起来很简单，但是应用却非常广泛。例如在排列组合问题中，很多场景的数学模型就是 Fibonacci 数列。下面是两个常见的例子。

1）楼梯问题

hdu 2041 题。有一楼梯共 M 级，刚开始时人在第一级，若每次只能跨上一级或两级，要走上第 M 级，共有多少种走法？

假设到第 n 级总共的走法为 $f(n)$。如何到达第 n 级？可以分成两种情况：①第一次跳 1 级，剩下 $n-1$ 个台阶，跳法是 $f(n-1)$；②第一次跳 2 级，剩下 $n-2$ 个台阶，跳法是 $f(n-2)$，所以 $f(n)=f(n-1)+f(n-2)$。这是一个 Fibonacci 数列。

2）矩形覆盖问题

用 2×1 的小矩形覆盖 $2n$ 的大矩形，总共有多少种方法？

假设方法总共有 $f(n)$，分成两种情况：①第一次放 1 格，剩下 $n-1$ 个格子，方法有 $f(n-1)$ 种；②第一次放 2 格，剩下 $n-2$ 个格子，方法有 $f(n-2)$ 种。这也是一个 Fibonacci 数列。

8.3.5 母函数

本节介绍一种求解递推关系的特殊思路——母函数。母函数（Generating Function，又译为生成函数）是算法竞赛中经常使用的一种解题方法，它用代数方法解决组合计数问题，是数学与应用的有趣结合。

本节尝试引导读者理解母函数，并用来解决一些算法问题。不过，读者仍然需要进一步深入地学习母函数的数学思想，这样才能更好地应用它。建议读者阅读一些组合数学方面的书籍，做一些习题，以加强理解[②]。

---

① 很多人说“黄金分割美学”其实是夸大其词。

② 《组合数学》，Richard A. Brualdi 著，冯舜玺译，机械工业出版社。

1. 整数划分

在讲解母函数之前先思考一个经典问题——整数划分。整数划分是指把一个正整数 n 分解成多个整数的和，这些数大于等于 1、小于等于 n。不同划分法的总数叫作划分数。例如 $n=4$ 时有 5 种划分，即{1,1,1,1}、{1,1,2}、{2,2}、{1,3}、{4}。

这个问题有很多扩展[①]，例如将 n 划分成最大数不超过 m 的划分数，$m\leqslant n$。当 $n=4$，$m=2$ 时有 3 种划分，即{1,1,1,1}、{1,1,2}、{2,2}。

> **hdu 1028 "Ignatius and the Princess III"**
>
> 求整数 n 有多少种划分，$1\leqslant n\leqslant 120$。
>
> 输入一个数字 n，输出划分数。

在引入母函数方法之前先用递归求解，代码如下，其中函数 part(n,n)返回对 n 划分的结果。

递归求整数划分

```
#include <bits/stdc++.h>
using namespace std;
int part(int n, int m) {                          //将 n 划分成最大数不超过 m 的划分数
    if(n == 1||m == 1)    return (1);
    else if(n < m)        return part(n,n);
    else if(n == m)       return 1 + part(n,n - 1);
    else        return part(n - m, m) + part(n, m - 1);  //这一行导致 TLE
}
int main(){
    int n;
    while(cin >> n)
     cout << part(n, n) << endl;
    return 0;
}
```

函数 part()的最后一行有两种情况。

(1) part($n-m$,m)：划分中有一个数为 m，那么从 n 中减去 m，继续对 $n-m$ 进行划分；

(2) part(n,$m-1$)：划分中每个数都小于 m，即每个数不大于 $m-1$，继续划分。

但是，用上面的递归代码提交 hdu 1028 题，结果是 TLE。观察程序的最后一行，发现递归翻倍，是 $O(2^n)$的复杂度。

用 DP 可以显著降低复杂度。把递归程序中的逻辑改写成递推式，在函数 part()中提前预计算出所有 n 的划分数。程序的计算复杂度是 $O(n^2)$。

用 DP 求整数划分

```
const int MAXN = 200;
int dp[MAXN + 1][MAXN + 1];            //dp[n][m]: 将 n 划分成最大数不超过 m 的划分数
```

① 扩展的划分问题：http://www.cnblogs.com/radiumlrb/p/5797168.html(永久网址：perma.cc/92XR-N9DF)。

```
void part() {                          //预计算 dp[n][m],求出所有 n 的划分
    for(int n = 1; n <= MAXN; n++)
        for(int m = 1; m <= MAXN; m++){
            if((n == 1)||(m == 1))     dp[n][m] = 1;
            else if(n < m)             dp[n][m] = dp[n][n];
            else if(n == m)            dp[n][m] = dp[n][m - 1] + 1;
            Else                       dp[n][m] = dp[n][m - 1] + dp[n - m][m];
        }
}
```

下面用母函数方法求解整数划分问题。

2. 母函数的概念

在解决整数划分问题之前先通过一个更简单的问题介绍母函数的概念。

问题：从数字 1、2、3、4 中取出一个或多个相加(每个数最多只能用一次)，能组合成几个数？每个数有几种组合？

在表 8.3 中，第 1 行是组合得到的数字，第 2 行是组合的情况，第 3 行是有几种组合。

视频讲解

表 8.3　数字组合问题

数字 S	1	2	3	4	5	6	7	8	9	10
组合	1	2	1+2 3	1+3 4	1+4 2+3	1+2+3 2+4	1+2+4 3+4	1+3+4	2+3+4	1+2+3+4
数量 N	1	1	2	2	2	2	2	1	1	1

下面引进一个公式，并把公式展开，这个公式能解决上面的数字组合问题。后文会介绍这个公式是怎么来的。

$$(1+x)(1+x^2)(1+x^3)(1+x^4)=1+x+x^2+2x^3+2x^4+2x^5+2x^6+2x^7+x^8+x^9+x^{10}$$

读者仔细观察，可以发现公式和上面的表是有关系的：

(1) 公式左边的 x 的幂与组合用到的数字 1、2、3、4 相对应。观察公式左边，包括 4 个部分，$(1+x)$中的 x 是 1 次幂，$(1+x^2)$中的 x^2 是 2 次幂，依此类推，刚好是数字 1、2、3、4。

(2) 公式右边 x 的幂与表格中的组合数 S 是对应的。公式右边 x 的幂从 1 到 10，组合数 S 也从 1 到 10。

(3) 公式右边的系数与表格中的数量 N 相对应，都是 1、1、2、2、2、2、2、1、1、1。

因此，用这个公式可以计算上面的组合数问题。

这就是母函数的原理：**“把组合问题的加法与幂级数的乘幂对应起来”**。

那么，这个公式是如何得到的？

为了更容易理解，把公式左边写成以下的形式：

$$(1+x)(1+x^2)(1+x^3)(1+x^4)$$
$$=(x^{0\times1}+x^{1\times1})(x^{0\times2}+x^{1\times2})(x^{0\times3}+x^{1\times3})(x^{0\times4}+x^{1\times4})$$

其含义以公式右边的$(x^{0\times1}+x^{1\times1})$为例，$x^{0\times1}$表示不用数字 1，$x^{1\times1}$表示用数字 1。

所以，这个公式实际上就是组合问题的反映：**用或者不用数字 1、用或者不用数字 2、用或者不用数字 3**，依此类推，公式就是这样构造出来的。公式构造出来后，把它展开后的结果就是组合问题的答案。

母函数的定义：对于序列 $a_0,a_1,a_2,\cdots$，构造函数 $G(x)=a_0+a_1x+a_2x^2\cdots$，称 $G(x)$ 为序列 $a_0,a_1,a_2,\cdots$的母函数。

简单地说，母函数是一种幂级数，其中每一项的系数反映了这个序列的信息。在本例中，a_kx^k 的 a_k 是数 k 的组合数量。

3. 用母函数解决整数划分问题

整数划分比上面的数字组合问题复杂一些，因为整数划分的数字是可以重复的。可以这样设计整数划分的母函数：

$$(x^{0\times1}+x^{1\times1}+x^{2\times1}+\cdots)(x^{0\times2}+x^{1\times2}+x^{2\times2}+\cdots)(x^{0\times3}+x^{1\times3}+x^{2\times3}+\cdots)\cdots$$
$$=(1+x+x^2+\cdots)(1+x^2+x^4+\cdots)(1+x^3+x^6+\cdots)\cdots$$

其中，$(x^{0\times1}+x^{1\times1}+x^{2\times1}+\cdots)$的含义是**不用数字 1、用一次 1、用两次 1**，依此类推。

母函数展开后，第 x^n 项的系数就是数字 n 的划分数。

那么，如何编程计算母函数展开后的系数？模拟手工计算过程就可以了。首先把前两部分$(1+x+x^2+\cdots)$和$(1+x^2+x^4+\cdots)$相乘并展开；展开的结果再与第 3 部分$(1+x^3+x^6+\cdots)$相乘并展开；继续这个过程直到完成。

用母函数求整数划分(hdu 1028)

```
const int MAXN = 200;
int c1[MAXN + 1], c2[MAXN + 1];
void part() {
   int i, j, k;
   for(i = 0; i <= MAXN; i++){             //初始化,即第 1 部分(1 + x + x² + … )的系数,都是 1
       c1[i] = 1;   c2[i] = 0;
   }
   for(k = 2; k <= MAXN; k++){             //从第 2 部分(1 + x² + x⁴ + … )开始展开
       for(i = 0; i <= MAXN; i++)
       //k = 2 时,i 循环第 1 部分(1 + x + x² + … ),j 循环第 2 部分(1 + x² + x⁴ + … )
           for(j = 0; j + i <= MAXN; j += k)
               c2[i + j]  += c1[i];
       for(i = 0; i <= MAXN; i++) {        //更新本次展开的结果
           c1[i]  = c2[i];  c2[i]  = 0;
       }
   }
}
```

数组第 c1[n]项用来记录每次展开后第 x^n 项的系数，计算结束后，c1[n]就是整数 n 的划分数。数组 c2[]用于记录临时计算结果。

上面的代码可以当成模板，请读者仔细理解细节。虽然不同的问题有不同的母函数，但都是方程式的展开，代码和上面的差不多，只要做相应的修改即可。

本节讲解的是“普通型”母函数，可用于求组合方案数；还有一种“指数型”母函数，用于求排列数。例如{1,2,3,4}，要求每个数字用且只用一次，那么组合方案只有 1 种，而排列有

4!=24 种。

求组合方案的题目,如果能用普通型母函数求解①,一般也能用 DP 求解。但是,众所周知,DP 的难点在于递推关系,想不到就做不出来;而母函数的思路是很直观的,容易理解。比如整数划分问题,母函数的方法要比 DP 简单一些。

4. 指数型母函数

先看一个典型的例题——hdu 1521 题。

> **hdu 1521 "排列组合"**
>
> 有 n 种物品,并且知道每种物品的数量,求从中选出 m 件物品的排列数。例如有两种物品 A、B,并且数量都是 1,从中选两件物品,则排列有"AB"和"BA"两种。
>
> 输入:每组输入数据有两行,第 1 行是两个数 n 和 m($1 \leqslant m, n \leqslant 10$),表示物品数;第 2 行有 n 个数,分别表示这 n 件物品的数量。
>
> 输出:对应每组数据输出排列数(任何运算不会超出 2^31 的范围)。

分析题目,假设有 3 种物品 A、B、C,数量分别是 2、3、1,即{A,A,B,B,B,C},从中选两件物品,则排列是{AA,AB,BA,AC,CA,BB,BC,CB},共 8 种。

针对这个例子,直接给出指数型母函数的解决方案。下面表达式的第 1 行是母函数公式,第 2 行展开,第 3 行整理:

$$
\begin{aligned}
G(x) &= \left(1+\frac{x}{1!}+\frac{x^2}{2!}\right)\left(1+\frac{x}{1!}+\frac{x^2}{2!}+\frac{x^3}{3!}\right)\left(1+\frac{x}{1!}\right) \\
&= 1+3x+4x^2+\frac{19}{6}x^3+\frac{19}{12}x^4+\frac{1}{2}x^5+\frac{1}{12}x^6 \\
&= 1+\frac{3x}{1!}+\frac{8x^2}{2!}+\frac{19}{3!}x^3+\frac{38}{4!}x^4+\frac{60}{5!}x^5+\frac{60}{6!}x^6
\end{aligned}
$$

第 1 行的 3 个括号内分别代表两个 A、3 个 B、1 个 C。

答案就隐含在最后一行中。例如$\frac{19}{3!}x^3$,x^3 的幂 3 表示选 3 件物品,系数 19 表示有 19 种排列。这一行给出了所有的答案:物品 A、B、C,数量分别有 2、3、1 个,那么选一件物品的排列有 3 种、选两件有 8 种、选 3 件有 19 种、选 4 件有 38 种、选 5 件有 60 种、选 6 件有 60 种。

是不是很神奇?下面分析母函数公式。

把公式写成$\frac{x}{1!}$、$\frac{x^2}{2!}$、$\frac{x^3}{3!}$这样的形式,实际上是在处理排列。例如,第 1 行的第 1 部分$\left(1+\frac{x}{1!}+\frac{x^2}{2!}\right)$是对物品 A(有两个 A)的排列。为了容易理解,可以改写成$\left(\frac{1x^0}{0!}+\frac{1x^1}{1!}+\frac{1x^2}{2!}\right)$,意思如下:

$\frac{1x^0}{0!}$,不选 A 的排列有 1 种,即{ϕ};

$\frac{1x^1}{1!}$,选 1 件 A 的排列有 1 种,即{A};

① 这里对整数划分给出了有趣的解释:《组合数学》,Richard A. Brualdi 著,机械工业出版社,8.3 节,分拆数。

$\frac{1x^2}{2!}$,选两件 A 的排列有 1 种,即{AA}。

同理,选 B 物品(有 3 个 B)时计算公式是$\left(1+\frac{x}{1!}+\frac{x^2}{2!}+\frac{x^3}{3!}\right)$,选 C 物品(有 1 个 C)时计算公式是$\left(1+\frac{x}{1!}\right)$。

当同时选多个物品时,把公式相乘,其展开项就是排列情况。例如选 A、B 两种物品,A 的$\frac{x}{1!}$与 B 的$\frac{x^3}{3!}$相乘,表示选 1 个 A,再选 3 个 B 的排列数量,计算得到$\frac{x}{1!}\times\frac{x^3}{3!}=\frac{x^4}{6}=\frac{4x^4}{4!}$,分子系数是 4,表示有 4 种排列,它们是{ABBB,BABB,BBAB,BBBA}。

为什么要将分母写成 1!、2!、3! 这样的形式?它体现了排列和组合的关系:k 个物品的排列和 k 个物品的组合相差 k! 倍。在选多个物品时,利用这个特点可以处理多重组合的排列问题。

例如 A 有两个、B 有 3 个,组合只有一种,是{A,A,B,B,B},下面求排列数。

(1) 两个 A 的排列公式是$\frac{x^2}{2!}$,分母的 2! 处理了排列的问题:如果是两个不同的 A_1、A_2,应该有两种排列,即{A_1A_2,A_2A_1},但是 A_1、A_2 相同,所以需要除以 2!,得到一种排列{AA}。

(2) 3 个 B 的排列公式是$\frac{x^3}{3!}$,分析是一样的,分母除以 3!,剔除重复的排列,得到一种排列{BBB}。

(3) 合起来排列公式是$\frac{x^2}{2!}\times\frac{x^3}{3!}=\frac{x^5}{12}=\frac{10x^5}{5!}$,分子的系数是 10,表示有 10 种排列{AABBB,ABABB,ABBAB,…}。

现在给出指数型母函数的定义:对序列 $a_0,a_1,a_2,\cdots$,构造函数 $G(x)=a_0+\frac{a_1}{1!}x+\frac{a_2}{2!}x^2+\frac{a_3}{3!}x^3\cdots$,称 $G(x)$ 为序列 $a_0,a_1,a_2,\cdots$的指数型母函数。

指数型母函数的程序和普通型母函数的程序非常相似,只多了对分母 k! 的处理。

hdu 1521 题的程序留给读者自己编写。

8.3.6 特殊计数

1. Catalan 数

1) 定义

Catalan 数是一个数列,它的一种定义如下:

$$C_n=\frac{1}{n+1}\binom{2n}{n},\quad n=0,1,2,\cdots$$

前一部分 Catalan 数是 1,1,2,5,14,42,132,429,1430,4862,16796,58786,208012,742900,2674440,9694845,35357670…Catalan 数的增长速度极快。

Catalan 数看起来有点奇怪,但是观察它的公式,其中有组合计数。实际上,Catalan 数

是很多组合计数应用问题的数学模型，是一个很常见的数列①。

Catalan 数有以下两种基本模型。

模型Ⅰ：$C_n=\frac{1}{n+1}\binom{2n}{n}=\binom{2n}{n}-\binom{2n}{n+1}=\binom{2n}{n}-\binom{2n}{n-1}$

其中，$\binom{2n}{n}$是在 $2n$ 种情况中选 n 个的组合数；$\binom{2n}{n-1}$是在 $2n$ 种情况中选 $n-1$ 个的组合数。注意，$\binom{2n}{n-1}$和$\binom{2n}{n+1}$等价。

模型Ⅰ的公式可以从一个**基本模型**推导出来：把 n 个 1 和 n 个 0 排成一行，使这一行的任意前 k 个数中 1 的数量总是大于或等于 0 的数量(或者 0 的数量大于等于 1 的数量，二者等价)。这样的排列有多少个？答案是这样的排列一共有 C_n 个，即 Catalan 数。

模型Ⅱ：第Ⅱ种模型是递推。

$$C_n = C_0C_{n-1} + C_1C_{n-2} + \cdots + C_{n-2}C_1 + C_{n-1}C_0 = \sum C_kC_{n-k}, \quad C_0 = 1$$

下面几个应用场景可以按上面两个模型进行解释。

2）棋盘问题

hdu 2067 题。

hdu 2067 "小兔的棋盘"

小兔的叔叔从外面旅游回来给它带来了一个礼物，小兔高兴地跑回自己的房间，拆开一看是一个棋盘，小兔有所失望。不过没过几天它发现了棋盘的好玩之处，从起点(0,0)走到终点(n,n)的最短路径数是 $C(2n,n)$，现在小兔想如果不穿过对角线(但可接触对角线上的格点)，这样的路径数有多少？

题目的意思是一个 n 行 n 列的棋盘，从左下角走到右上角，一直在对角线右下方走，不穿过主对角线，走法有多少种？例如 $n=4$ 时有 14 种走法。

这个问题就是上面的基本模型(Ⅰ)，下面进行分析。

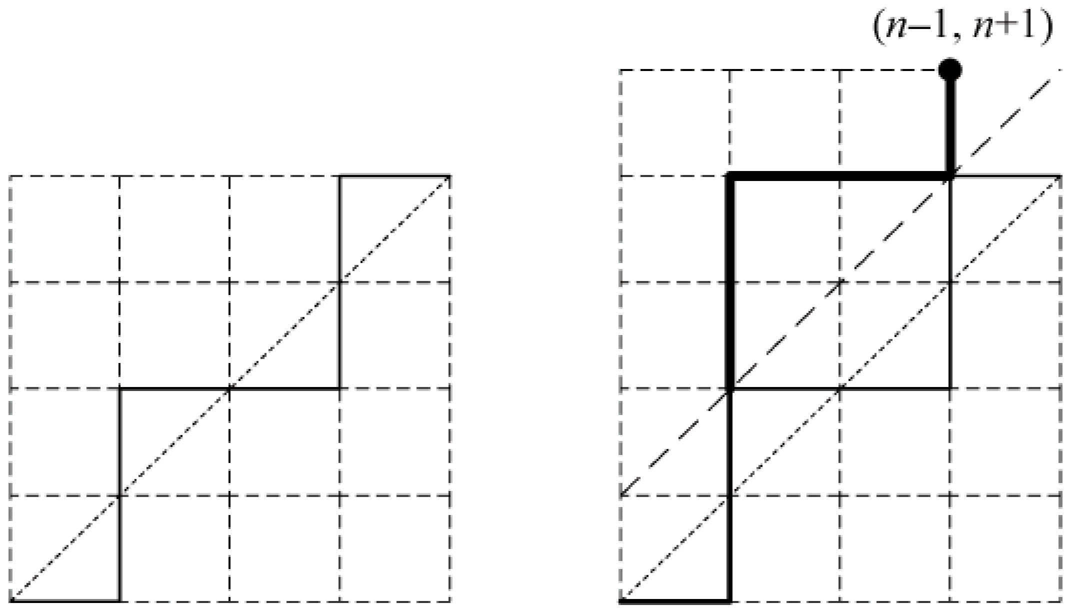
对方向编号，向上是 0，向右是 1，那么从左下角走到右上角一定会经过 n 个 1 和 n 个 0。满足要求的路线是走到任意一步 k，前 k 步中向右的步数(1 的个数)大于或等于向上的步数(0 的个数)，否则就穿过对角线了。

设从左下角走到右上角的总路线有 X 条，分成 3 个部分：对角线下面的 A 条路线，对角线上面的 B 条路线，穿过对角线的 C 条路线。不过，这 3 个部分可以简化为两个部分，即对角线下面的 A、穿过对角线的 Y(包括 B 和 C)。$A=X-Y$ 就是答案。

总路线 $X=\binom{2n}{n}$，它的意思是在 $2n$ 个位置放 n 个 1(剩下的 n 个肯定是 0)，这样的数有$\binom{2n}{n}$个。

① 这里列出了很多 Catalan 数的应用，注意看其中的棋盘问题：https://en.wikipedia.org/wiki/Catalan_number (短网址：t.cn/R1TgbAG)。

对于 Y，需要用到一种叫作 André's reflection method 的方法。图 8.1(a)给出了一条穿过对角线的路线(即 C 路线；或者给出一条在斜对角上方并不穿过对角线的路线，即 B 路线，分析和 C 路线一样)。在图 8.1(b)中，画一条新的对角线，把它画在原来对角线的上面一格。



(a) 一条穿过对角线的路线　　(b) 按新对角线映射

图 8.1　André's reflection method

下面开始操作：原来的路线，从左下角出发，第一次接触到这条新对角线后，把剩下的部分以新对角线为轴进行映射，得到新的路线。这条新的路线即图 8.1 中加粗的黑线。加粗黑线下面的一部分黑线是原来的，保持不变；上面一部分是新的，与原来那一部分对称。整个路线仍然是连续的，但是路线的终点变为$(n-1,n+1)$。注意，"在原对角线右下方不穿过主对角线的走法"，即前文提到的 A 部分，与新对角线无交集，无法映射，被排除在外。

新的路线和原来的路线是一一对应的。这些新路线有多少个？此时有 $n+1$ 个 0、$n-1$ 个 1，共 $2n$ 个；选出 $n-1$ 个 1(等价于选出 $n+1$ 个 0)的排列有 $Y=\binom{2n}{n-1}$个。

因此 $A=X-Y=\binom{2n}{n}-\binom{2n}{n-1}$。

3) 括号问题

括号问题：用 n 个左括号和 n 个右括号组成一串字符串有多少种合法的组合？例如，"( ) ( ) (( ))"是合法的，而"( )) (( )"是非法的。显然，合法的括号组合是：任意前 k 个括号组合，左括号的数量大于等于右括号的数量。

定义左括号为 0、右括号为 1。问题转化为 n 个 0 和 n 个 1 组成的序列，在任意前 k 个序列中 0 的数量都大于等于 1 的数量。模型和上面的棋盘问题一样。

读者可以练习 hdu 5184 题：给定初始的括号序列，再给定 n 表示序列的总长度，问一共有多少种括号组成方式？

4) 出栈序列问题

给定一个以字符串形式表示的入栈序列，求出一共有多少种可能的出栈顺序？比如入栈序列为{1 2 3}，则出栈序列一共有 5 种，即{1 2 3}、{1 3 2}、{2 1 3}、{2 3 1}、{3 2 1}。

分析可知，合法的序列是对于出栈序列中的每一个数字，在它后面的比它小的所有数字一定是按递减顺序排列的。例如，{3 2 1}是合法的，3 出栈之后，比它小的后面的数字是{2 1}，且这个顺序是递减顺序；而{3 1 2}是不合法的，因为在 3 后面的数字{1 2}是一个递

增的顺序。

对于每一个数来说，必须进栈一次、出栈一次。定义进栈操作为 0、出栈操作为 1。n 个数的所有状态对应 n 个 0 和 n 个 1 组成的序列。出栈序列，即要求进栈的操作数大于等于出栈的操作数。问题转化为由 n 个 1 和 n 个 0 组成的 $2n$ 位二进制数，任意前 k 个序列中 0 的数量大于或等于 1 的数量。结果仍然是 Catalan 数。

hdu 1023 题：火车进站、出站，模拟进栈和出栈操作。由于要计算第 100 个 Catalan 数，这个数非常大，需要用大数计算，读者可以用 Java 编程。

5) 二叉树问题

n 个结点构成的二叉树共有多少种情况？

例如有 3 个结点(图中的黑点)的二叉树，可以构成 5 种二叉树，如图 8.2 所示。

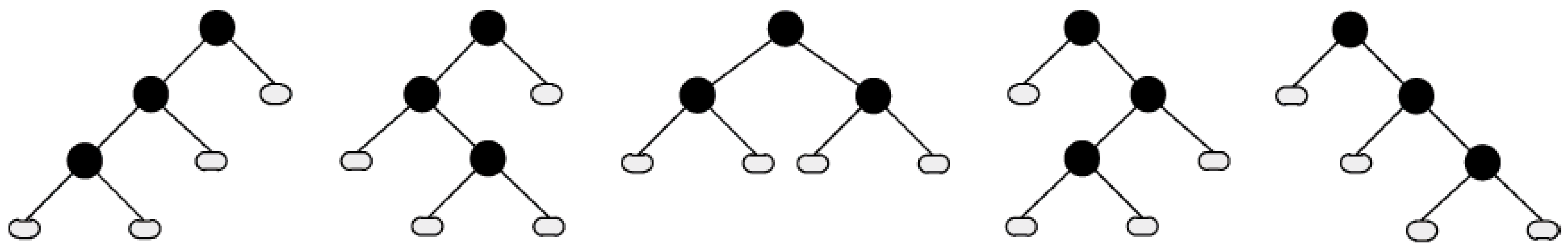


图 8.2　包括 3 个结点的二叉树

这个问题符合模型Ⅱ：

$$C_n = C_0C_{n-1} + C_1C_{n-2} + \cdots + C_{n-2}C_1 + C_{n-1}C_0 = \sum C_kC_{n-k}, \quad C_0 = 1$$

其含义如下：

C_0C_{n-1}：右子树有 0 个结点＋左子树有 $n-1$ 个结点；

C_1C_{n-2}：右子树有 1 个结点＋左子树有 $n-2$ 个结点；

⋮

$C_{n-1}C_0$：右子树有 $n-1$ 个结点＋左子树有 0 个结点。

读者可以练习 hdu 1130/3240 题。

6) 其他问题

买票找零问题。

三角剖分问题：把一个凸多边形内部划分成多个三角形有多少种方法？

7) 编程计算 Catalan 数

有多种计算方法：

(1) $C_n = C_0C_{n-1} + C_1C_{n-2} + \cdots + C_{n-2}C_1 + C_{n-1}C_0 = \sum C_kC_{n-k}, \quad C_0 = 1$

(2) $C_n = \dfrac{4n-2}{n+1}C_{n-1}, \quad C_0 = 1$

(3) $C_n = \dfrac{1}{n+1}\dbinom{2n}{n} = \dfrac{(2n)!}{(n+1)!n!}$

从公式(2)可知，当 n 很大时，$C_n/C_{n-1} \approx 4$。所以 Catalan 数是以约 4^n 递增的，增长极快。

这 3 个公式的应用场合不同。

用公式(1)的场合：需要输出 Catalan 数的值。此时 n 较小，例如算 $n \leqslant 100$ 内的 Catalan 数，不过 Catalan 数仍然是一个超级大的数。此时用公式(1)比用公式(2)好。因为公式(2)需要算大数的乘/除法，它比公式(1)的递推公式更容易溢出。例如 hdu 2067 题“小

兔的棋盘”，读者可以分别用两种方法编程。可以发现，如果只是简单地用 int64 来定义 Catalan 数，当计算到第 34 个 Catalan 数时公式(2)计算出错，而公式(1)仍然正确。对于更大的 Catalan 数，需要进行高精度计算，例如 hdu 1023/1130 题，计算第 100 个 Catalan 数。

用公式(2)、(3)的场合：n 非常大，不能直接输出 Catalan 数，而是做取模操作。例如 hdu 5184，需要算第 10 万个 Catalan 数，用公式(1)算太慢了。此时用公式(2)是很好的选择。不过，(2)和(3)都有大数除法，对大数做除法会损失精度，所以需要转换为逆元，然后再取模。如果用公式(3)算，注意先预计算 n 的阶乘(算阶乘的同时对阶乘取模)，然后再用公式计算。

【习题】

除了上面的基础题外，读者可练习下面的题目：

hdu 4828，卡特兰数，逆元。

hdu 5673，卡特兰数，逆元。

hdu 5177，$n \leqslant 10^{18}$ 的卡特兰数。

2. Stirling 数

Stirling 数也是解决特定组合问题的数学工具，包括两种，即第一类 Stirling 数和第二类 Stirling 数，它们有相似的地方。

首先通过一个经典的仓库钥匙问题来了解第一类 Stirling 数。

问题描述：有 n 个仓库，每个仓库有两把钥匙，共 $2n$ 把钥匙，有 n 位保管员。

问题 1：如何放钥匙使得保管员都能够打开所有仓库?

问题 2：保管员分别属于 k 个不同的部，部中的保管员数量和他们管理的仓库数量一样多，例如第 i 个部有 m 个管理员，管 m 个仓库。如何放钥匙，使得同部的所有保管员能打开本部的所有仓库，但是无法打开其他的仓库?

问题 1 很好解答。1 号仓库放 2 号仓库的钥匙，2 号仓库放 3 号仓库的钥匙，依此类推，n 号仓库放 1 号仓库的钥匙，相当于 n 个仓库形成了一个闭环的圆；然后每个保管员拿一把钥匙即可，他打开一个仓库后就能拿到下一把钥匙，继续打开其他所有的仓库。

问题 2 是问题 1 的扩展：把 n 个仓库分成 k 个圆排列，每个圆内部按问题 1 处理。这里的麻烦问题是：把 n 个仓库分配到 k 个圆里，不能有空的圆，共有多少种分法? 答案就是第一类 Stirling 数。

1) 第一类 Stirling 数

定义第一类 Stirling 数 $s(n,k)$：把 n 个不同的元素分配到 k 个圆排列里，圆不能为空。问有多少种分法?

下面直接给出第一类 Stirling 数的递推公式①：

$$s(n,k) = s(n-1,k-1) + (n-1)s(n-1,k), \quad 1 \leqslant k \leqslant n$$

$$s(0,0) = 1, \quad s(k,0) = 0, \quad 1 \leqslant k \leqslant n$$

① 《组合数学》，Richard A. Brualdi 著，机械工业出版社。第 8 章，定理 8.2.9，推导了第一类 Stirling 数的递推公式。

根据递推公式计算部分 Stirling 数，如表 8.4 所示。

表 8.4　第一类 Stirling 数 $s(n,k)$的值

n \ k	0	1	2	3	4	5	6	…
0	1							
1	0	1						
2	0	1	1					
3	0	2	3	1				
4	0	6	11	6	1			
5	0	24	50	35	10	1		
6	0	120	274	225	85	15	1	
…								

例如：

$s(2,1)=1$，两个物体 a、b 放在 1 个圆圈里，有 1 种方案，即{(ab)}；

$s(3,1)=2$，3 个物体 a、b、c 放在 1 个圆圈里，有两种方案，即{(abc)}和{(acb)}；

$s(3,2)=3$，3 个物体 a、b、c 放在两个圆圈里，有 3 种方案，即{(ab),(c)}、{(ac),b}、{(a),(bc)}。

2) 第二类 Stirling 数

定义第二类 Stirling 数 $S(n,k)$：把 n 个不同的球分配到 k 个相同的盒子里①，不能有空盒子。问有多少种分法？

$S(n,k)$的递推公式如下：

$$S(n,k)=kS(n-1,k)+S(n-1,k-1),\quad 1\leqslant k\leqslant n$$
$$S(0,0)=1,\quad S(i,0)=0,\quad 1\leqslant i\leqslant n$$

根据递推公式计算部分 Stirling 数，如表 8.5 所示。

表 8.5　第二类 Stirling 数 $S(n,k)$的值

n \ k	0	1	2	3	4	5	6	…
0	1							
1	0	1						
2	0	1	1					
3	0	1	3	1				
4	0	1	7	6	1			
5	0	1	15	25	10	1		
6	0	1	31	90	65	15	1	
…								

① 读者自然能想到，根据球是否一样、盒子是否相同、盒子是否可为空可以组合成各种类似的问题，例如把 n 个一样的球分配到 k 个相同的盒子里、把 n 个一样的球分配到 k 个不同的盒子里，等等。在这些问题中，第二类 Stirling 数比较复杂，但它是很基本的问题。所有的情况参考《应用组合数学》，Fred S. Roberts、Barry Tesman 著，冯速译，机械工业出版社，2.10 节，分装问题；公式的推导见 5.5.3 节。

例如：

$S(2,1)=1$，两个球 a、b 放在 1 个盒子里，有 1 种方案，即{(ab)}；

$S(3,1)=1$，3 个球 a、b、c 放在 1 个盒子里，有 1 种方案，即{(abc)}；

$S(3,2)=3$，3 个球 a、b、c 放在两个相同的盒子里，有 3 种方案，即{(ab),(c)}、{(ac),b}、{(a),(bc)}。

【习题】

hdu 4372 "Count the Buildings"，第一类 Stirling 数。

hdu 2643 "Rank"，第二类 Stirling 数。

8.4 概率和数学期望

概率和数学期望是概率论和统计学中的数学概念。设有随机变量 X，出现取值 x_i 的概率是 p_i，把它们的乘积之和称为数学期望(Expected Value，或者均值 mean)，记为 $E(X)$：

$$E(X)=\sum_{i=1}^{n}x_ip_i$$

$E(X)$是基本的数学特征之一，它反映了随机变量平均值的大小。

以妇女的生育率为例，假设某国有 2000 万个育龄妇女，不生育妇女有 277 万，一孩 724 万，二孩 883 万，三孩 116 万。记一个妇女的孩子数量是 X，取值 0、1、2、3，概率分别是 277/2000=0.1385、724/2000=0.362、883/2000=0.4415、116/2000=0.058。那么平均每个妇女生育的孩子数量如下：

$$E(X)=0\times0.1385+1\times0.362+2\times0.4415+3\times0.058=1.419$$

数学期望具有线性性质。有限个随机变量之和的数学期望等于每个变量的数学期望之和：

$$E(X+Y)=E(X)+E(Y)$$

竞赛中求数学期望的题目一般都会用到它的线性性质。由于线性性质和 DP 的状态转移思想很相似，所以常常用 DP 来实现。

1. 例题 1

首先看一个简单的例题。

poj 2096 "Collecting Bugs"

一个软件有 s 个子系统，会产生 n 种 bug。现在要找出所有种类的 bug。假设某人一天发现一个 bug。一个 bug 属于某个子系统的概率是 $1/s$，属于某种分类的概率是 $1/n$。问发现 n 种 bug，且每个子系统都发现 bug 的天数的期望。$0<n,s\leqslant1000$。

输入：n 和 s；

输出：数学期望。

输入样例：
1 2
输出样例：
3.0000

定义状态 dp[i][j]，它表示已经找到 i 种 bug，并存在于 j 个子系统中，要达到目标状态还需要的期望天数。其中，dp[n][s]表示已经找到 n 种 bug，且存在于 s 个子系统，说明已经达到了目标，还需要 0 天，所以 dp[n][s]=0。从 dp[n][s]倒推回 dp[0][0]，就是本题的答案，即还没有找到任何 bug 的情况下到达 dp[n][s]时需要的期望天数。

从 dp[i][j]开始：后面 1 天找到 1 个 bug，可能有以下 4 种情况。

(1) dp[i][j]：发现一个 bug，属于已经有的 i 个分类和 j 个系统，概率为 $p1=(i/n)*(j/s)$。这一天相当于浪费了。

(2) dp[$i+1$][j]：发现一个 bug，不属于已有分类、属于已有系统，概率为 $p2=(1-i/n)*(j/s)$。

(3) dp[i][$j+1$]：发现一个 bug，属于已有分类、不属于已有系统，概率为 $p3=(i/n)*(1-j/s)$。

(4) dp[$i+1$][$j+1$]：发现一个 bug，不属于已有系统、不属于已有分类，概率 $p4=(1-i/n)*(1-j/s)$。

可以验证：p1+p2+p3+p4=1。

状态转移方程如下：

$$\begin{aligned}\mathrm{dp}[i][j] = {} & \mathrm{p1} * \mathrm{dp}[i][j] + \mathrm{p2} * \mathrm{dp}[i+1][j] + \mathrm{p3} * \mathrm{dp}[i][j+1] + \\ & \mathrm{p4} * \mathrm{dp}[i+1][j+1] + 1 \qquad // \text{末尾加上 1 天}\end{aligned}$$

整理得到：

$$\begin{aligned}\mathrm{dp}[i][j] &= (\mathrm{p2} * \mathrm{dp}[i+1][j] + \mathrm{p3} * \mathrm{dp}[i][j+1] + \mathrm{p4} * \mathrm{dp}[i+1][j+1] + 1)/(1-p1) \\ &= (n*s + (n-i)*j*\mathrm{dp}[i+1][j] + i*(s-j)*\mathrm{dp}[i][j+1] + \\ &\quad (n-i)*(s-j)*[i+1][j+1])/(n*s-i*j)\end{aligned}$$

在写程序时，从 dp[n][s]倒推到 dp[0][0]，dp[0][0]就是答案。

poj 2096 部分程序

```
cin >> n >> s;
for (int i = n; i >= 0; i-- )
  for (int j = s; j >= 0; j-- ){
    if ( i == n && j == s )
      dp[n][s] = 0.0;
    else
      dp[i][j] = (n * s + (n - i) * j * dp[i + 1][j] + i * (s - j) * dp[i][j + 1]
                + (n - i) * (s - j) * dp[i + 1][j + 1]) /(n * s - i * j);
  }
```

2. 例题 2

hdu 4035 是经典的迷宫概率问题，综合了图、数学期望、DP 等内容。

hdu 4035 "Maze"

一个迷宫有 n 个房间，用 $n-1$ 条隧道连通起来。每个房间里都有陷阱和逃生口。某人的起点在房间 1，在每个房间都有 3 种可能：

（1）落入陷阱被杀死，回到房间 1，概率为 k_i；

（2）找到逃生口，走出迷宫，概率为 e_i；

（3）在该房间连接的隧道中随机走一条，进入下一个房间。

求逃出迷宫所要走的隧道数量的期望值。

首先分析这个迷宫，它是一棵树。

一个有 n 个点、$n-1$ 条边的无向连通图，图上肯定没有回路，这样的图是一棵树。证明如下：用反证法，假设有一个回路，那么在这个回路上可以删除一条边而不影响整体的连通；删除之后，还有 n 个点、$n-2$ 条边，这是不可能连通的。

要使有 n 个点的图是连通的，至少需要 $n-1$ 条边。生成一个连通图，可以用扩大路径法，从一个点开始，每加入一个新的点，至少需要一条边来连接，所以 n 个点至少需要 $n-1$ 条边才能连通。

下面是推导过程和编程思路。

1）定义 DP 状态 $E[i]$

在结点 i 处，逃出迷宫所要走的边数的期望。

$E[1]$就是所求的答案。

根据树的特点，分析 $E[i]$：

i 是叶子结点，即 i 没有子结点。在结点 i 有 3 种情况，即被杀、逃出、回到父结点。

$$E[i] = k_i * E[1] + e_i * 0 + (1 - k_i - e_i) * (E[\text{father}[i]] + 1) \tag{8-1}$$

i 是非叶子结点，设 i 连接的边数是 m，有 3 种情况，即被杀、逃出、转到其他结点。

$$E[i] = k_i * E[1] + e_i * 0 + (1 - k_i - e_i)/m * (E[\text{father}[i]] + 1 + \sum(E[\text{child}[i]] + 1)) \tag{8-2}$$

2）计算过程

设对于每个结点：

$$E[i] = A_i * E[1] + B_i * E[\text{father}[i]] + C_i$$

目标是求 $E[1]$，$E[1]=A_1 * E[1]+B_1 * 0+C_1$，即 $E[1]=C_1/(1-A_1)$。

在叶子结点上有：

$$A_i = k_i$$
$$B_i = 1 - k_i - e_i$$
$$C_i = 1 - k_i - e_i$$

在非叶子结点上，设 j 为 i 的子结点，则：

$$\begin{aligned}\sum(E[\text{child}[i]]) &= \sum E[j] \\ &= \sum(A_j * E[1] + B_j * E[\text{father}[j]] + C_j) \\ &= \sum(A_j * E[1] + B_j * E[i] + C_j)\end{aligned}$$

代入式(8-1)和式(8-2)中,消去 $E[\text{child}[i]]$ 和 $E[\text{father}[j]]$,可以得到非叶子结点的 A_i、B_i、C_i 的表达式。

3) 编程思路

从上面的推导过程可知,计算过程是从叶子结点开始算,再算它们的父结点,直到算出根结点的 A_1、B_1、C_1,得到 $E[1]$。在编程时,需要按从叶子结点到根结点的顺序遍历每个点。

这个过程用 DFS 编程是最合适的。从根结点 1 出发,用 DFS 遍历整棵树;DFS 到最底层的叶子结点时,计算叶子结点的 A_i、B_i、C_i,然后逐步回退,再计算非叶子结点的 A_i、B_i、C_i。

在题目中,图的规模 $n \leqslant 10\,000$,需要用邻接表存储。请读者在学习第 10 章的相关内容后再回头做这一题。

【习题】

hdu 3853 “LOOPS”,基础题。

hdu 4405 “Aeroplane chess”,简单题。

poj 3071 “Football”,简单概率 DP。

poj 3744 “Scout YYF I”,用矩阵优化求概率。

hdu 4089 “Activation”,2011 年北京区域赛题目,概率 DP,较难。

8.5 公平组合游戏

本节讨论的公平组合游戏(Impartial Combinatorial Game,ICG)[1]是满足以下特征的一类问题:

视频讲解

(1) 有两个玩家,游戏规则对两人是公平的;

(2) 游戏的状态有限,能走的步数也有限;

(3) 两人轮流走步,当一个玩家不能走步时游戏结束;

(4) 游戏的局势不能区分玩家身份,像围棋这样有黑、白两方的游戏就不属于此类问题。

ICG 问题有一个特征:给定初始局势,并且指定先手玩家,如果双方都采取最优策略,那么获胜者就已经确定了。也就是说,ICG 问题存在必胜策略。

本节讲解 ICG 问题的必胜策略,有关的知识点有 P-position、N-position、Nim Game、Sprague-Grundy 函数、威佐夫游戏等。ICG 很早就得到了研究,例如对于 Nim Game 问题,1902 年 C. Bouton 在一本著作中进行了分析;对于 Sprague-Grundy 函数,由数学家 Grundy 和 Sprague 在 1930 年分别独立发现。

---

① 在算法竞赛中,常常称这类问题是“博弈论”问题。虽然 Nim Game、Sprague-Grundy 函数也属于博弈论的范畴,不过在普通的博弈论教材中并不能找到有关内容。在一些应用组合数学书中会提到有关知识,请参考《应用组合数学》,Alan Tucker 著,冯速译,人民邮电出版社,第 11 章。

Sprague-Grundy 函数是本节最重要的内容。

8.5.1 巴什游戏与 P-position、N-position

首先给出一个简单的例题。小学奥数中有这样的题目。

1. 巴什游戏(Bash Game)

> **hdu 1846 "Brave Game"**
>
> 有 n 颗石子,甲先取,乙后取,每次可以拿 $1\sim m$ 颗石子,轮流拿下去,拿到最后一颗的人获胜。
>
> 输入: n 和 m, $1\leqslant n,m\leqslant 1000$。
>
> 输出:如果先拿的甲赢了,输出"first",否则输出"second"。

程序非常简单,若 $n\%(m+1)==0$,则先手败,否则先手胜。

```
cin >> n >> m;
if(n % (m+1) == 0)     printf("second\n");
else                   printf("first\n");
```

分析如下:

(1) 当 $n\leqslant m$ 时,由于一次最少拿 1 个、最多拿 m 个,甲可以一次拿完,先手赢。

(2) 当 $n=m+1$ 时,无论甲拿走多少个($1\sim m$ 个),剩下的都多于 1 个、少于等于 m 个,乙都能一次拿走剩余的石子,后手取胜。

上面两种情况可以扩展为以下两种情况:

(Ⅰ) 如果 $n\%(m+1)=0$,即 n 是 $m+1$ 的整数倍,那么不管甲拿多少,例如 k 个,乙都拿 $m+1-k$ 个,使得剩下的永远是 $m+1$ 的整数倍,直到最后的 $m+1$ 个,所以后拿的乙一定赢。

(Ⅱ) 如果 $n\%(m+1)!=0$,即 n 不是 $m+1$ 的整数倍,还有余数 r,那么甲拿走 r 个,剩下的是 $m+1$ 的倍数,这样就转移到了情况(Ⅰ),相当于甲、乙互换,结果是甲赢。

在这个拿石子的游戏里,对于后拿的乙来说是很不利的,只有在 $n\%(m+1)=0$ 的情况下乙才能赢,在其他情况下都是甲赢。

2. P-position、N-position 与动态规划

上面对巴什游戏的解答虽然很好理解,但是如果稍作扩展,就不那么容易了。例如取石子的数量,不是 $1\sim m$ 内的连续数字,而是只能在 $\{a_1,a_2,\cdots,a_k\}$ 中选。对于此类问题,有必要研究一种通用的方法。

定义 P-position 为前一个玩家(Previous Player,即刚走过一步的玩家)的必胜位置、N-position 为下一个玩家(Next Player)的必胜位置。

当前状态是 N-position,表示马上走下一步的先手必胜;P-position 表示先手必败。

设只能拿数量为 $\{1,4\}$ 的石头。在表 8.6 中,x 是石头的数量,pos 是对应的 position。

表 8.6 只能拿数量为{1,4}的石头

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	…
pos	P	N	P	N	N	P	N	P	N	N	P	N	P	N	…

表中的 pos 是这样计算的：

(1) $x=0,1,2,3,4$ 时，pos=P,N,P,N,N。特别注意 $x=0$，即没有石头的情况，可以看成下一个玩家(先手玩家)没有石头可拿，输了，pos=P。$x=1$ 时，先手玩家必赢，pos=N。$x=2$ 时，先手只能拿 1 个，后手拿剩下的 1 个，后手赢，pos=P。

(2) $x=5$ 时分两种情况：如果先手玩家拿 1 个，退回到 $x=5-1=4$ 的情况，此时后手玩家处于 N，即后手处于赢的位置；如果先手拿 4 个，退回到 $x=5-4=1$ 的情况，此时后手仍然处于 N。在两种情况下后手都赢了。所以 $x=5$ 时，pos=P，即先手必输。

(3) $x=6$ 时，分别退回到 $x=6-1=5$ 和 $x=6-4=2$ 的情况，后手都处于 P。在两种情况下，后手都输了。所以 $x=6$ 时 pos=N，先手必赢。

(4) $x=7$ 时略。

(5) $x=8$ 时：退回到 $x=8-1=7$，后手处于 P；退回到 $x=8-4=4$，后手处于 N。在后手有输有赢的情况下，先手肯定选让对方必败的方案，所以 $x=8$ 时 pos=N。

可以观察到 pos 值是周期性变化的，周期为 5。

下面再举一个例子，设只能拿数量为{1,3,4}的石头，请读者验证表 8.7。

表 8.7 只能拿数量为{1,3,4}的石头

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
pos	P	N	P	N	N	N	N	P	N	P	N	N	N	N	P

pos 仍然是周期变化的，周期是 7。

上面的计算过程符合**动态规划**的思路。在编程时可以用动态规划，也可以直接按周期性变化规律做求余计算，hdu 1846 是一种最简单的情况，用求余编程计算就可以了。

巴什游戏有一些变形。例如 hdu 2147 "kiki's game"，给出一个 $n\times m$ 的矩阵，从右上角走到左下角，看谁先到终点。画出 P-N 图，找到规律即可。

8.5.2 尼姆游戏

巴什游戏只有一堆石头，如果扩展到多堆石头，情况将复杂得多，这就是尼姆游戏(Nim Game)[①]。

尼姆游戏的规则：有 n 堆石子，数量分别是$\{a_1,a_2,a_3,\cdots,a_n\}$，两个玩家轮流拿石子，每次从任意一堆中拿走任意数量的石子，拿到最后一个石子的玩家获胜。

以 3 堆石头为例，简单情况的胜负如下。

$\{0,0,0\}$、$\{0,1,1\}$、$\{0,k,k\}$：先手必败。

$\{1,1,1\}$、$\{1,1,2\}$、$\{1,1,3\}$：先手必胜。

① https://en.wikipedia.org/wiki/Nim。

对于任意的$\{a_1,a_2,a_3,\cdots,a_n\}$，尼姆游戏有一个极为简单的判断胜负的方法，即做异或运算。

定理 8.1：

若 $a_1 \oplus a_2 \oplus a_3 \oplus \cdots \oplus a_n \neq 0$，则先手必胜，记此时的状态为 N-position；

若 $a_1 \oplus a_2 \oplus a_3 \oplus \cdots \oplus a_n = 0$，则先手必败，记此时的状态为 P-position。

例如 3 堆石头的数量分别是{5,7,9}，转化为二进制数后做异或运算，结果如下：

$$\begin{array}{c} 0101 \\ 0111 \\ 1001 \\ \hline 1011 \end{array}$$

异或运算的结果不等于 0，先手必胜。

在数学中，二进制的异或运算也可以看成是统计每一位上 1 的总个数的奇偶性：如果这一位上有偶数个 1，那么这一位的计算结果为 0；如果有奇数个，计算结果为 1。所以，尼姆游戏中的异或运算也被称为 **Nim-sum 运算**。

下面对定理 8.1 做简单的证明。

（1）必定能够从 N-position 转化到 P-position。也就是说，先手处于必胜点 N-position 时可以拿走一些石子，让后手必败。读者可以先自己思考如何转化。下面是具体方法：任选一堆，例如第 i 堆，石头数量是 k；对剩下的 $n-1$ 堆做异或运算，设结果为 H；如果 H 比 k 小，就把第 i 堆石头减少到 H；这样操作之后，因为 $H \oplus H = 0$，所以 n 堆石头的异或等于 0。可以证明，总会存在这样的第 i 堆石头，而且可能有多种转化方案。下面例题 hdu 1850 的程序中的“if((sum ^a[i])<=a[i])”统计了所有方案。

（2）进入 P-position 后，轮到的下一个玩家，不管拿多少石子都会转移到 N-position。因为任何一堆的数量变化，都会使得这一堆的二进制数至少有一位发生变化，导致异或运算的结果不等于 0。也就是说，这一个玩家不管怎么拿石子都必败。

（3）在游戏过程中，按上述（1）和（2）的步骤在 N-position 和 P-position 之间交替转化，直到所有堆的石头都是 0，即终止于 P-position。

上述证明过程也说明了玩家该如何进行游戏。

hdu 1850 “Being a Good Boy in Spring Festival”

两人小游戏：桌子上有 n 堆扑克牌；每堆牌的数量分别为 a_i；两人轮流进行；每走一步可以从任意一堆中取走任意张牌；桌子上的扑克牌全部取光，则游戏结束；最后一次取牌的人为胜者。问先手的人如果想赢，第一步有几种选择？

输入：n 表示扑克牌的堆数；$a_i(i=1\sim n)$表示每堆扑克牌的数量。

输出：如果先手能赢，输出他第一步可行的方案数，否则输出 0。

主要代码如下：

```
int sum = 0, ans = 0;                        //sum 是 Nim - sum,ans 是第一步可行的方案数
for(int i = 0; i<n; i++)   sum ^ = a[i];     //异或计算,求 Nim - sum
```

```
    if(sum == 0)    cout << 0 << endl;              //开始局面是 P-position,先手必败
    else{                                           //开始局面是 N-position,先手胜
      for(int i = 0; i < n; i++)
        if((sum ^a[i]) <= a[i])                     //计算第一步所有的可能方案
          ans++;
      cout << ans << endl;
    }
```

程序中的“if((sum ^a[i]) <= a[i])”计算第一步的方案数,它利用了异或运算的原理:$A \oplus B \oplus B = A$。设 H 等于除了 $a[i]$之外其他所有数的异或,有:

$$\text{sum} = H \,\hat{}\, a[i]$$

$$\text{sum} \,\hat{}\, a[i] = H \,\hat{}\, a[i] \,\hat{}\, a[i] = H$$

所以,(sum ^$a[i]$)<=$a[i]$就是 H<=$a[i]$。把 $a[i]$减少到 H,就是一种可行的方案。

8.5.3　图游戏与 Sprague-Grundy 函数

前面讲解的巴什游戏、尼姆游戏用 P-position 和 N-position 做分析工具,如果遇到更复杂的游戏,很难分析。有一种高级的分析方法,即 Sprague-Grundy 函数,是巴什游戏、尼姆游戏这类问题的通用方法,该方法用图作为分析工具。

图游戏的规则是:给定一个有向无环图,在一个起点上放一枚棋子,两个玩家交替将这枚棋子沿有向边进行移动,无法移动者判负。图是有向无环图的,不会有环路,保证游戏有终点。

像巴什游戏、尼姆游戏这样的 ICG 问题都可以转化为基于图的游戏。把 ICG 中的每个局势看成图上的一个结点,在每个局势和它的后继局势之间连一条有向边,就抽象成了图游戏。下面给出图游戏的严格定义。

1. 图游戏

定义:一个有向无环图 $G(X,F)$,X 是点(局势)的非空集合,F 是 X 上的函数,对于 $x \in X$,有 $F(x) \subset X$;对于给定的 $x \in X$,$F(x)$表示玩家从 x 出发能够移动到的位置;如果 $F(x)$为空,说明无法继续移动,称 x 是终点位置。

两个玩家的游戏过程按以下规则进行:一个玩家先走,起点是 x_0,然后两人交替走步;在位置 x,玩家可以选择移动到 y 点,$y \in F(x)$;位于终点位置的玩家,判负。

例如在巴什游戏中,设一次可以拿的石头是{1,2},结点集合是 $X=\{0,1,2,\cdots,n\}$。$F(0)$为空,因为石子数量是 0,已经到达终点,无法再转移;$F(1)=\{0\}$,表示从 1 可以转移到 0;$F(2)=\{0,1\}$,表示从 2 可以转移到 0 或 1;等等。这里以 $n=6$ 为例画出游戏图,如图 8.3 所示。

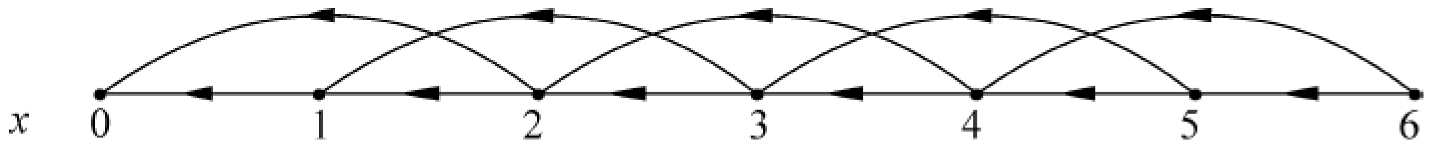


图 8.3　巴什游戏图

图 8.3 中的每个点表示一个可能的局势,箭头表示局势的转移方向。玩家的所有步骤都在这个图上。图上有一些是先手必胜点(N-position),例如 1、2、4、5 等,以及先手必败点

(P-position),例如 3、6 等。确定了这些关键的点,就能得到解决方案。

但是,在大部分情况下游戏图是很复杂的,例如尼姆游戏,给定 3 堆石头{5,7,9},图上的每个点是一个局势,如{0,0,0}、{0,1,1}等,可能的局势有 $6\times8\times10=480$ 个,点与点之间的转移关系也很复杂。

利用 Sprague-Grundy 函数这个工具可以轻松地找到这些关键点。

2. Sprague-Grundy 函数

定义:在一个图 $G(X,F)$中,把结点 x 的 Sprague-Grundy 函数定义为 sg(x),它等于没有指定给它的任意后继结点的 sg 值的最小非负整数。

上述定义有些拗口,下面的例子清晰地说明了它的含义。图 8.3 中每个结点的 sg 值如图 8.4 所示。

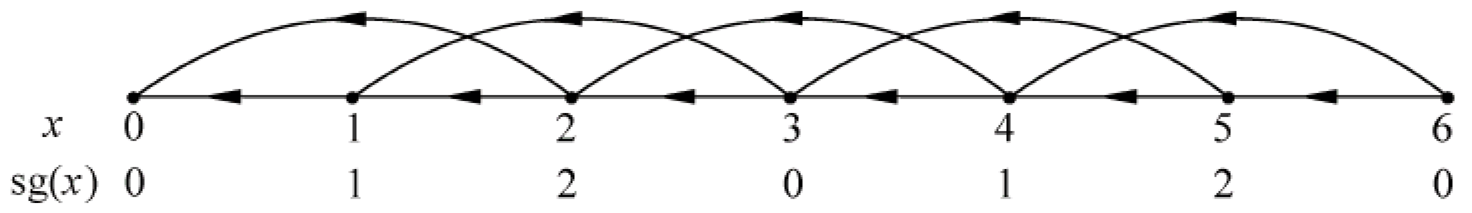


图 8.4 结点 x 和 sg(x)

当 $x=0$ 时,sg(0)=0,因为结点 0 没有后继结点,0 是最小的非负整数;

当 $x=1$ 时,结点 1 的后继是结点 0,由于 sg(0)=0,不等于 sg(0)的最小非负整数是 1,所以 sg(1)=1;

当 $x=2$ 时,结点 2 的后继是结点 0 和 1,由于 sg(0)=0、sg(1)=1,不等于 sg(0)和 sg(1)的最小非负整数是 2,所以 sg(2)=2;

当 $x=3$ 时,结点 3 的后继是结点 1 和 2,由于 sg(1)=1、sg(2)=2,不等于 sg(1)和 sg(2)的最小非负整数是 0,所以 sg(3)=0;

当 $x=4$ 时,结点 4 的后继是结点 2 和 3,由于 sg(2)=2、sg(3)=0,不等于 sg(2)和 sg(3)的最小非负整数是 1,所以 sg(4)=1;

等等。

上面的说明也给出了求每个点的 sg 值的过程,和前面提到的用动态规划思路求 P-position、N-position 的过程差不多,复杂度是 $O(nm)$,其中 n 是石子数量,m 是一次最多可拿的石子数。

3. 用 Sprague-Grundy 函数求解巴什游戏

在只有一堆石子的巴什游戏中,以下判断成立:

sg(x)=0 的结点 x 是必败点,即 P-position 点。

证明如下:

(1) 根据 sg 函数的性质,有以下推论:sg(x)=0 的结点 x,没有 sg 值等于 0 的后继结点;sg(y)>0 的任意结点 y,必有一条边通向 sg 值为 0 的某个后继结点。

(2) 如果 sg(x)=0 的结点 x 是图上的终点(没有后继结点,在图论中称这个点的出度为 0),显然有 $x=0$,它是一个 P-position 点;如果 x 有后继结点,那么这些后续结点都能通向某个 sg 值为 0 的结点。当玩家甲处于 sg(x)=0 的结点时,它只能转移到 sg(x)≠0 的结点,下一个玩家乙必然转移到 sg(x)=0 的点,从而再次让甲处于不利的局势。所以 sg(x)=0

的点是必败点。

仍然以 hdu 1846 为例，用 Sprague-Grundy 函数的方法编程实现。

hdu 1846 程序（sg 函数）

```
#include<bits/stdc++.h>
using namespace std;
const int MAX = 1001;
int n, m, sg[MAX], s[MAX];
void getSG(){
    memset(sg, 0, sizeof(sg));
    for (int i=1; i<=n; i++){
        memset(s, 0, sizeof(s));
        for (int j=1; j<=m && i-j>=0; j++)
            s[sg[i-j]] = 1;                    //把 i 的后继结点放到集合 s 中
        for (int j=0; j<=n; j++)               //计算 sg[i]
            if(!s[j]){sg[i]=j; break;}
    }
}
int main(){
    int c;  cin>>c;
    while (c--){
        cin>>n>>m;
        getSG();
        if (sg[n])  cout<<"first\n";           //sg != 0,先手胜
        else        cout<<"second\n";          //sg == 0,后手胜
    }
    return 0;
}
```

4. 用 Sprague-Grundy 函数求解尼姆游戏

尼姆游戏中有多堆石头，也可以用 Sprague-Grundy 函数求解。其步骤如下：

(1) 计算每一堆石头的 sg 值；

(2) 求所有石头堆的 sg 值的异或，其结论是：

若 $sg(x_1)\oplus sg(x_2)\oplus sg(x_3)\oplus\cdots\oplus sg(x_n)\neq 0$，先手必胜；

若 $sg(x_1)\oplus sg(x_2)\oplus sg(x_3)\oplus\cdots\oplus sg(x_n)=0$，先手必败。

请读者根据前面对尼姆游戏的说明以及 Sprague-Grundy 函数的特征证明其正确性。

下面用 Sprague-Grundy 函数求解 hdu 1848。

hdu 1848 "Fibonacci again and again"

两人小游戏，定义如下：一共有 3 堆石子，数量分别是 m、n、p；两人轮流走；每走一步可以选择任意一堆石子，然后取走 f 个；f 只能是菲波那契数列中的元素（即每次只能取 1、2、3、5、8 等数量）；最先取光所有石子的人为胜者。

输入：3 个整数 m、n、p（$1\leqslant m,n,p\leqslant 1000$），$m=n=p=0$ 表示输入结束。

输出：如果先手的人能赢，输出"Fibo"，否则输出"Nacci"。

这一题属于典型的尼姆游戏，程序如下：

hdu 1848 程序(sg 函数)

```
#include<bits/stdc++.h>
using namespace std;
const int MAX = 1001;
int sg[MAX], s[MAX];
int fibo[15] = {1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987};
void getSG(){                                    //计算每一堆的 sg 值
    for(int i = 0;i <= MAX;i++){
        sg[i] = i;
        memset(s, 0, sizeof(s));
        for(int j = 0; j < 15 && fibo[j]<= i; j++){
            s[sg[i - fibo[j]]] = 1;
            for(int j = 0; j <= i; j++)
                if(!s[j]) {sg[i] = j; break;}
        }
    }
}
int main(){
    getSG();                                     //预计算 sg 值
    int n,m,p;
    while(cin >> n >> m >> p && n + m + p){
        if(sg[n]^sg[m]^sg[p])  cout << "Fibo" << endl;
        else                   cout << "Nacci"<< endl;
    }
    return 0;
}
```

【习题】

hdu 1907 "John",尼姆游戏。
hdu 2999 "Stone Game, Why are you always there?",sg 函数。
hdu 1524 "A Chess Game",sg 函数。
hdu 4111 "Alice and Bob",sg 函数,记忆化搜索。
hdu 4203 "Doubloon Game",数据规模大,找规律。

8.5.4 威佐夫游戏

威佐夫游戏(Wythoff's Game)是一种结论非常有趣的游戏,其原型见 hdu 1527 的描述。

> **hdu 1527 "取石子游戏"**
>
> 有两堆石子,数量任意,可以不同。游戏开始由两个人轮流取石子。
>
> 游戏规定每次有两种不同的取法,一是可以从任意的一堆中取走任意多的石子;二是可以从两堆中同时取走相同数量的石子。最后把石子全部取完者为胜者。
>
> 现在给出初始的两堆石子的数目 a 和 b,问先手玩家是不是最后的胜者?

分析两堆石子的数量(a,b)，使先手必输的局势有(0,0)、(1,2)、(3,5)、(4,7)、(6,10)、(8,13)、(9,15)，等等，称这些局势为“奇异局势”[①]。

观察发现，奇异局势有两个特征：①差值是递增的，分别是0,1,2,3,4,…；②每个局势的第一个值是未在前面出现过的最小的自然数。经过分析可以发现，每个奇异局势的第一个值总是等于这个局势的差值乘上黄金分割比例1.618，然后取整。

需要注意的是，在推导奇异局势时用到的黄金分割数需要较高的精度，直接用1.618这个估值是不行的。在程序中，用以下公式计算高精度黄金分割数：

```
double gold = (1 + sqrt(5))/2;
```

下面是hdu 1527的代码。

hdu 1527 代码

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    int n, m;
    double gold = (1 + sqrt(5))/2;                //黄金分割 = 1.618 033 98…
    while(cin >> n >> m){
        int a = min(n, m), b = max(n, m);
        double k = (double)(b - a);
        int test = (int)(k * gold);               //乘以黄金分割数，然后取整
        if(test == a)  cout << 0 << endl;         //先手败
        else           cout << 1 << endl;         //先手胜
    }
    return 0;
}
```

8.6 小 结

数学题是算法竞赛中的重点内容，包含的内容也相当广泛。本章讲解了一些竞赛中基本的和常用的知识点，还有很多大类没有涉及，例如积分、线性规划、傅里叶变换等。

有一些比较基础的知识点本章没有提到，但是需要读者掌握，例如高斯消元、中国剩余定理、Polya原理、欧拉函数、莫比乌斯函数等。

在一个竞赛队中，所有队员都需要掌握本章的内容，并且至少应该有一个队员深入钻研数学类题目。

① 威佐夫游戏的奇异局势和黄金分割数有关，Fibonacci数列也和黄金分割数有关，两者在这里发生了联系。关于威佐夫游戏的奇异局势和黄金分割数之间关系的证明，请参考“http://www.matrix67.com/blog/archives/6784”(永久网址：perma.cc/BCX4-9XXJ)。

第9章　字　符　串

✍常用字符串函数
✍字符串哈希
✍字典树
✍KMP
✍AC 自动机
✍后缀树和后缀数组

字符串处理是竞赛中的常见题目，除了简单的字符串查找、替换、匹配等问题以外，还有比较复杂的字符串算法，其中应用广泛的有字符串哈希、KMP、字典树(Trie Tree)、AC 自动机和后缀数组等。

9.1　字符串的基本操作

字符串的基本操作有读入、查找、替换、截取、数字和字符串转换等。下面用一个例题介绍字符串的读入、查找和替换操作。

poj 3981 “字符串替换”

读取一个字符串，把其中所有的"you"替换成"we"。

下面的程序一次读取一个完整的字符串，用 gets()函数实现。

C 程序 1

```
#include <stdio.h>
char str[1002];
int main(){
  int i;
  while(gets(str) != NULL) {
    for(int i = 0; str[i]!= '\0'; i++)
        if(str[i] == 'y' && str[i + 1] == 'o' && str[i + 2] == 'u') {
           printf("we");
           i += 2;
        }
        else
           printf("%c",str[i]);
    printf("\n");
  }
```

```
  return 0;
}
```

下面的程序一次只读一个字符，用 getchar()函数实现。这个程序比上一个程序要好，因为它不需要定义一个字符串数组，当然也不用考虑数组的大小。

C 程序 2

```
#include<stdio.h>
int main(void){
    char ch1, ch2, ch3;
    while((ch1 = getchar()) != EOF) {
    if(ch1 == 'y') {
      if((ch2 = getchar()) == 'o')  {
          if((ch3 = getchar()) == 'u')
             printf("we");
          else
             printf("yo%c",ch3);
      }
      else
          printf("y%c",ch2);
    }
    else
      putchar(ch1);
}
return 0;
}
```

下面的程序用到 string 类、getline()函数。

C++ 程序

```
#include<bits/stdc++.h>
using namespace std;
int main(){
   string str;
   int pos;
   while(getline(cin, str)){
      while((pos = str.find("you")) != -1)
         str.replace(pos, 3, "we");
      cout << str << endl;
   }
   return 0;
}
```

【习题】

hdu 1062，字符串反转。

hdu 6013，字符串反转，尺取法。

hdu 5007,子串查找。

hdu 1238,求多个字符串的最大公共子串,用暴力法做。

hdu 4054,输出字符的 ASCII 码。

hdu 2055,字符串和数字转换。

hdu 5938,字符串和数字转换。

9.2 字符串哈希

首先看一个比较特殊的字符串匹配问题:在很多字符串中尽快操作某个字符串。如果字符串的规模很大,访问速度很关键,具体例子参考 hdu 2648 题。在本书第 3 章的“3.1.7 map”中曾以 hdu 2648 为例讲解了用 map 容器匹配字符串的方法。这里用字符串哈希的方法重新编程处理。

这个问题用哈希(hash)方法解决是最快的。用哈希函数对每个子串进行哈希,分别映射到不同的数字,即一个整数哈希值,然后就可以根据哈希值找到子串,接下来配合使用数据结构或 STL 完成判重、统计、查询等操作。

哈希函数是其中的核心。理论上,任意函数 $h(x)$ 都可以是哈希函数,不过一个好的哈希函数应该尽量避免冲突。这个字符串哈希函数最好是完美哈希函数。完美哈希函数是指没有冲突的哈希函数:把 n 个子串的 key 值映射到 m 个整数上,如果对任意的 key1≠key2,都有 $h(\text{key1}) \neq h(\text{key2})$,这就是完美哈希函数。此时必然有 $n \leqslant m$。更进一步,如果 $n=m$,称为最小完美哈希函数。

那么如何找到一个接近完美的字符串哈希函数?有一些经典的字符串哈希函数,例如 BKDRHash、APHash、DJBHash、JSHash 等。一般使用 BKDRHash,求得的哈希值几乎不会冲突碰撞。但在实际应用时由于得到的哈希值都很大,不能直接映射到一个巨大的空间上,所以一般需要限制空间。方法是取余:把得到的哈希值对一个设定的空间大小取余数,以余数作为索引地址。当然,这样做会产生冲突问题。

下面用字符串哈希方法重新求解 hdu 2648。

在下面的程序中,哈希函数 BKDRHash()计算字符串的 hash 值,返回一个 unsigned int 数。根据上面的讨论可知,由于这个数可能很大,不能直接分配空间,程序用一个较小的 N 取余,分配到大小为 N 的空间。这样做会产生冲突,所以程序的大部分代码是解决冲突问题。

视频讲解

hdu 2648 的字符串哈希程序

```
#include<bits/stdc++.h>
using namespace std;
const int N = 10005;
struct node{
    char name[35];
    int price;
};
vector<node> List[N];                          //用于解决冲突
```

```
unsigned int BKDRHash(char * str)  {              //哈希函数
   unsigned int seed = 31,key = 0;
   while( * str)
       key = key * seed + ( * str++);
   return key & 0x7fffffff;
}
int main(){
   int n, m, key, add, memory_price, rank, len;
   int p[N];
   char s[35];
   node t;
   while(cin >> n){
      for(int i = 0; i < N; i++)
          List[i].clear();
      for(int i = 0;i < n;i++){
          cin >> t.name;
          key = BKDRHash(t.name) % N;             //计算 hash 值,并求余
          List[key].push_back(t);                 //hash 值可能冲突,把冲突的哈希值都存起来
      }
      cin >> m;
      while(m -- ){
          rank = len = 0;
          for(int i = 0; i < n; i++){
              cin >> add >> s;
              key = BKDRHash(s) % N;              //计算 hash 值
              for(int j = 0; j < List[key].size(); j++)     //处理冲突问题
                 if(strcmp(List[key][j].name, s) == 0){
                    List[key][j].price += add;
                    if(strcmp(s,"memory") == 0)
                       memory_price = List[key][j].price;
                    else
                       p[len++] = List[key][j].price;
                    break;
                 }
          }
          for(int i = 0; i < len; i++)
             if(memory_price < p[i])
                rank++;
          cout << rank + 1 << endl;
      }
   }
   return 0;
}
```

【习题】

hdu 4821 “String”。

hdu 4080 “Stammering Aliens”。

hdu 4622 “Reincarnation”。

hdu 4622,字符串哈希,较难。

9.3 字 典 树

再次回顾一个常见的字符串匹配问题：在 n 个字符串中查找某个字符串。

如果用暴力的方法，需要逐个匹配每个字符串，复杂度是 $O(nm)$，m 是字符串的平均长度。这个操作的效率十分低。

那么有没有很快的方法？大家都有查英语字典的经验，例如查找单词“dog”，先翻到字典的 d 部分，再翻到第 2 个字母 o、第 3 个字母 g，一共找 3 次即可。查找任意单词，查找次数最多只需要这个单词的字母个数。

字典树就是模拟这个操作的数据结构，它的时间复杂度和空间复杂度都很好。

(1) 时间复杂度：插入和查找单词的复杂度都是 $O(m)$，其中 m 是待插入/查询字符串的长度。

(2) 空间复杂度：有公共前缀的单词只需要存一次公共前缀，节省了空间。

图 9.1 所示为单词 be、bee、may、man、mom、he 的字典树。

从图 9.1 可以归纳出字典树的基本性质：根结点不包含字符，除根结点外的每个子结点都包含一个字符；从根结点到某一个结点，路径上经过的字符连接起来，为该结点对应的字符串；每个结点的所有子结点包含的字符互不相同。

通常在实现的时候会在结点设置一个标志，标记该结点是否为单词的末尾，例如图中画线的字符。

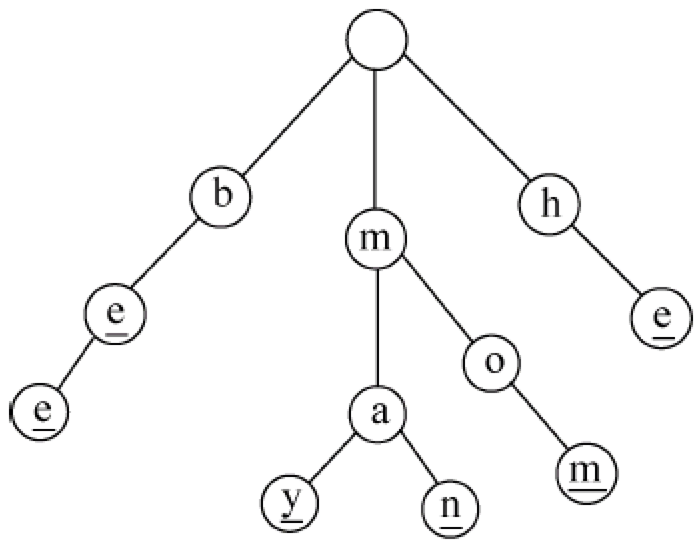


图 9.1 字典树

字典树有以下常见的应用：

(1) 字符串检索。检索、查询功能是字典树的基本功能。

(2) 词频统计。统计一个单词出现了多少次。

(3) 字符串排序。在插入的时候，在树的平级按字母表的顺序插入。字典树建好之后，用先序遍历，就得到了字典树的排序。

(4) 前缀匹配。字典树是按公共前缀来建树的，很适合用于搜索提示。例如 Linux 的行命令，输入一个命令的前面几个字母，系统会自动补全命令后面的字符。

字典树在本书“9.5 AC 自动机”中也有应用。

下面的例题给出了字典树的具体实现。

> **hdu 1251 “字典树”**
>
> 很多单词只由小写字母组成，不会有重复的单词出现，统计出以某个字符串为前缀的单词数量。

该题有多种方法。

1. 用 map 实现

这一题用 map 来做非常简单，代码如下：

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    char str[10];
    map<string, int> m;
    while(gets(str)){
        int len = strlen(str);
        if (!len)  break;                      //输入了一个空行
        for(int i = len; i>0; i--){
            str[i] = '\0';                     //从后往前删除这个字符串的字符,得到前缀
            m[str]++;                          //统计前缀的数量
        }
    }
    while(gets(str))  cout << m[str] << endl;
    return 0;
}
```

2. 用字典树实现

首先用正规的字典树实现,定义字典树的数据结构,并用指针指向下一层子树,代码很清晰。不过,由于本题的空间要求较高,Insert()内用 new Trie 分配的空间超过了题目的限制,代码会 MLE。

空间超额(MLE)的代码

```
#include<bits/stdc++.h>
using namespace std;
struct Trie{                                   //字典树的定义
    Trie* next[26];
    int num;                                   //以当前字符串为前缀的单词的数量
    Trie() {                                   //构造函数
       for(int i=0;i<26;i++) next[i] = NULL;
       num=0;
    }
};
Trie root;
void Insert(char str[]){                       //将字符串插入到字典树中
    Trie *p = &root;
    for(int i=0;str[i];i++){                   //遍历每一个字符
        if(p->next[str[i]-'a']==NULL)          //如果该字符没有对应的结点
           p->next[str[i]-'a'] = new Trie;     //创建一个
        p = p->next[str[i]-'a'];
        p->num++;
    }
}
int Find(char str[]){                          //返回以字符串为前缀的单词的数量
    Trie *p = &root;
    for(int i=0;str[i];i++){                   //在字典树中找到该单词的结尾位置
        if(p->next[str[i]-'a']==NULL)
           return 0;
        p = p->next[str[i]-'a'];
    }
```

```
        return p->num;
    }
    int main(){
        char str[11];
        while(gets(str)){
            if (!strlen(str))  break;              //输入了一个空行
            Insert(str);
        }
        while(gets(str))  cout << Find(str) << endl;
        return 0;
    }
```

更好、更紧凑的存储方法是用数组来实现字典树的数据结构，在竞赛中用这种方法更加保险。相关代码如下：

用数组实现字典树

```
int trie[1000010][26];                //用数组定义字典树，存储下一个字符的位置
int num[1000010] = {0};               //以某一字符串为前缀的单词的数量
int pos = 1;                          //当前新分配的存储位置
void Insert(char str[]){              //在字典树中插入某个单词
    int p = 0;
    for(int i = 0;str[i];i++){
        int n = str[i] - 'a';
        if(trie[p][n] == 0)           //如果对应字符还没有值
           trie[p][n] = pos++;
        p = trie[p][n];
        num[p]++;
    }
}
int Find(char str[]){                 //返回以某个字符串为前缀的单词的数量
    int p = 0;
    for(int i = 0;str[i];i++){
        int n = str[i] - 'a';
        if(trie[p][n] == 0)
           return 0;
        p = trie[p][n];
    }
    return num[p];
}
```

9.4 KMP

KMP 是单模匹配算法，即在一个长度为 n 的文本串中查找一个长度为 m 的模式串。它的复杂度是 $O(m+n)$，差不多是此类算法能达到的最优复杂度。

视频讲解

1. 朴素的模式匹配算法

在前面讲字符串哈希时曾用哈希解决了特定字符子串的匹配问题，下面

讨论更一般性的问题。

模式匹配(Pattern Matching):在一篇长度为 n 的文本 S 中,找某个长度为 m 的关键词 P。P 可能多次出现,都需要找到。这个一般性问题用哈希算法不合适,很麻烦。

最优的模式匹配算法复杂度能达到多好?由于至少需要检索文本 S 的 n 个字符和关键词 P 的 m 个字符,所以复杂度至少是 $O(m+n)$。

先考虑暴力方法(即朴素的模式匹配算法):在 S 的所有字符中逐个匹配 P 的每个字符。例如,S="abcxyz123",P="123"。第 1 次匹配,$P[0] \neq S[0]$,后面的 $P[1]$、$P[2]$就不用比较了。一共比较 6+3=9 次就好了,其中前 6 次对比 P 的第 1 个字符,第 7 次对比 P 的 3 个字符,如图 9.2 所示。

这个例子比较特殊,P 和 S 的字符基本上都不一样。在每次匹配时,往往第 1 个字符就对不上,用不着继续匹配 P 后面的字符。复杂度差不多是 $O(n+m)$,这已经是字符串匹配能达到的最优复杂度了。所以,如果字符串 S、P 符合这个特征,**用暴力法是不错的选择**。

但是,如果情况比较坏,例如 P 的前 $m-1$ 个都容易找到匹配,只有最后一个不匹配,那么复杂度就退化成 $O(nm)$。例如 S= "aaaaaaaab",P="aab",需要尝试 6×3+3=21 次,如图 9.3 所示,远远超过上面例子中的 9 次。

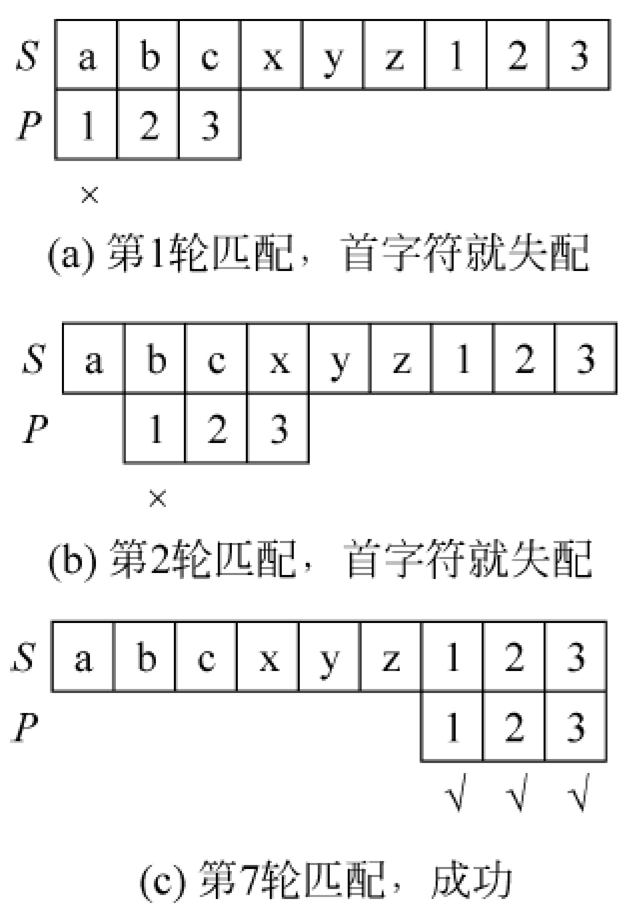


图 9.2　匹配示意

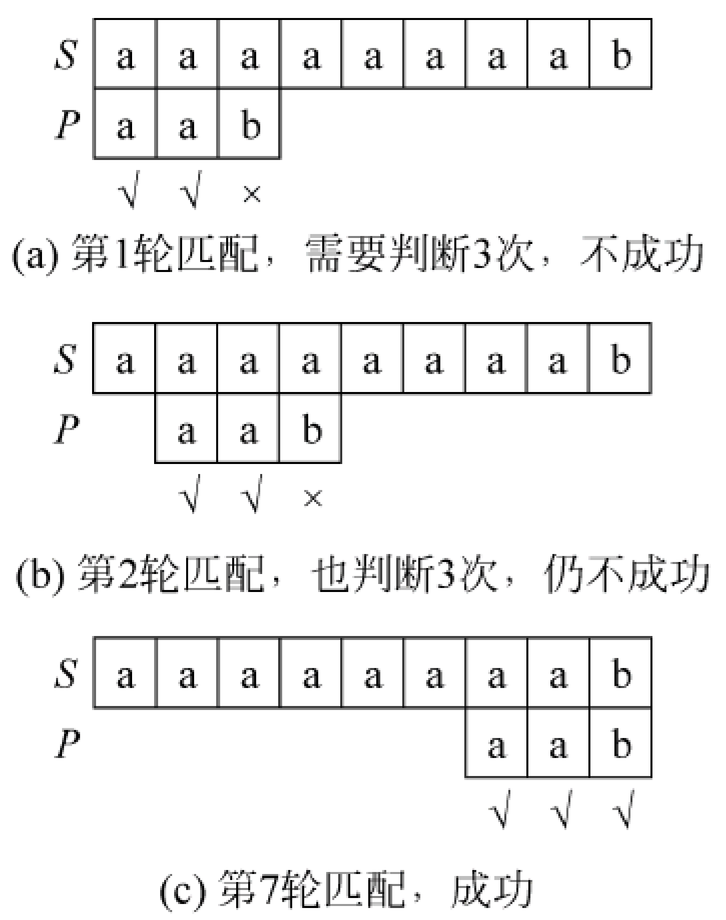


图 9.3　情况比较坏时的匹配

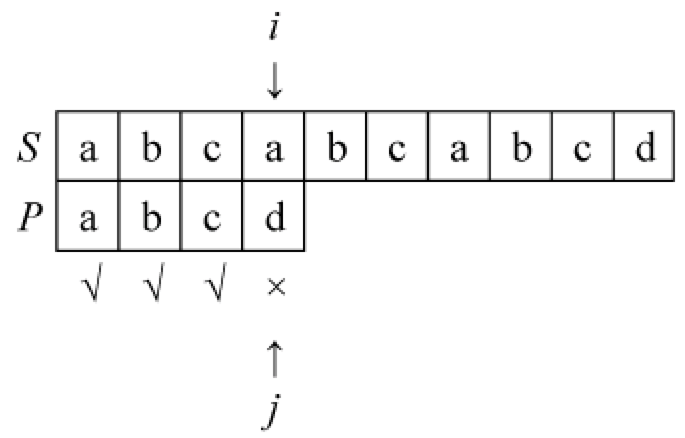
2. KMP 算法

KMP 是一种在任何情况下都能达到 $O(n+m)$复杂度的算法。它是如何做到的?简单地说,它通过分析 P 的特征对 P 进行预处理,从而在与 S 匹配的时候能够跳过一些字符串,达到快速匹配的目的。

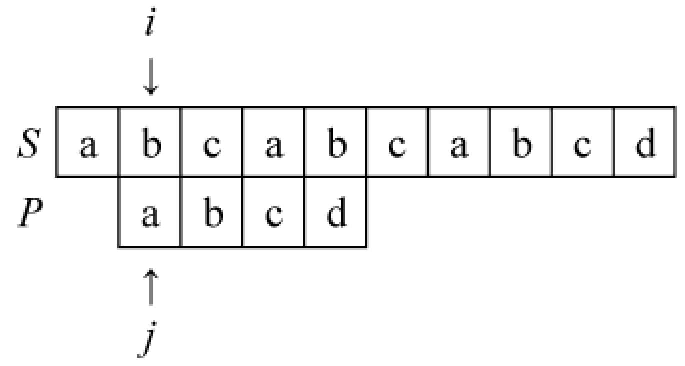
下面简单图解 KMP 的操作过程,如图 9.4 所示。$S[]$="abcabcabcd",$P[]$="abcd"。图中的 i 指向 $S[i]$,j 指向 $P[j]$,$0 \leqslant i < n$,$0 \leqslant j < m$。

图 9.4(c)说明,在用 KMP 算法时,指向 S 的 i 指针不会回溯,而是一直往后走到底。与图 9.4(b)的朴素方法相比,大大减少了匹配次数。请读者自己分析复杂度是否为 $O(n+m)$。

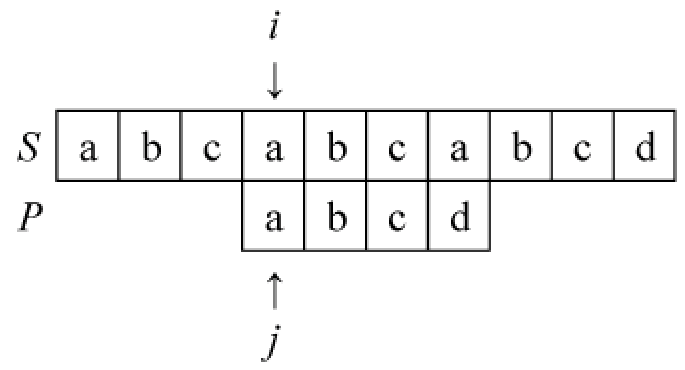
那么 KMP 是如何让 i 不回溯,只回溯 j 的呢?这就是 KMP 的核心——Next[]数组(也有写成 shift 或者 fail 的)。当出现失配后,进行下一次匹配时,用 Next[]指出 j 回溯的位置。



(a) 第1轮匹配后，在i=3，j=3的位置失配



(b) 第2轮匹配，如果用朴素方法，i和j回到i=1，j=0的位置重新开始



(c) 第2轮匹配，采用KMP算法，i=3不变，j回到j=0的位置重新开始

图 9.4　简单图解 KMP 的操作过程

Next[]是通过对 P 进行预处理得到的。在下面 hdu 2087 题的程序中用 getFail()函数求 Next[]数组。该程序虽然很短，却复杂难解，请读者自己阅读资料[①]。

有了 Next[]数组，就能很容易地写出 KMP 程序，代码见下面的例子。

3. KMP 模板题

hdu 2087“剪花布条”

一块花布条，上面印有一些图案，另有一块直接可用的小饰条，也印有一些图案。对于给定的花布条和小饰条，计算一下能从花布条中尽可能剪出几块小饰条。

输入：每一行是成对出现的花布条和小饰条。＃表示结束。

输出：输出能从花纹布中剪出的小饰条的最多个数。

输入样例：

abcde a3

aaaaaa　aa

＃

输出样例：

0

3

① “从头到尾彻底理解 KMP”，网址为“https://blog.csdn.net/v_july_v/article/details/7041827（永久网址：perma.cc/FY2G-6P67）”。

本题可以完全套用 KMP 的模板。KMP 算法的模板有两部分，即 getFail()和 kmp()。getFail()预计算 Next[]数组；kmp()函数实现在 S 中找 P，注意每次匹配到的起始位置是 $s[i+1-\text{plen}]$，末尾是 $s[i]$。

找到的匹配可能有很多个，而且可能重合，例如"aaaaaa"中包含了 3 个"aa"。但在本题中需要找到能分开的子串，即剪出不同的小饰条。这个问题容易解决，只需要在程序中加一句 if(i−last >= plen)进行判断即可。

KMP 程序

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 1000 + 5;
char str[MAXN], pattern[MAXN];
int Next[MAXN];
int cnt;
int getFail(char *p, int plen){
                  //预计算 Next[],用于在失配的情况下得到 j 回溯的位置
    Next[0] = 0; Next[1] = 0;
    for(int i = 1; i < plen; i++){
        int j = Next[i];
        while(j && p[i] != p[j])   j = Next[j];
        Next[i + 1] = (p[i] == p[j]) ? j + 1 : 0;
    }
}
int kmp(char *s, char *p) {                          //在 S 中找 P
   int last = -1;
    int slen = strlen(s), plen = strlen(p);
    getFail(p, plen);                                //预计算 Next[]数组
    int j = 0;
    for(int i = 0; i < slen; i++) {                  //匹配 S 和 P 的每个字符
        while(j && s[i]!= p[j])  j = Next[j];        //失配了,用 Next[]找 j 的回溯位置
        if(s[i] == p[j])  j++;                       //当前位置的字符匹配,继续
        if(j == plen) {                              //完全匹配
                //这个匹配,在 S 中的起点是 i + 1 - plen,末尾是 i,如有需要可以打印
                //printf("at location = %d, %s\n", i + 1 - plen,&s[i + 1 - plen]);
            //--------------------下面是与本题相关的工作
            if(i - last >= plen) {                   //判断新的匹配和上一个匹配是否能分开
                cnt++;
                last = i;                            //last 指向上一次匹配的末尾位置
            }
            //--------------------
        }
    }
}
int main(){
    while(~scanf("%s", str)){                        //读串
        if(str[0] == '#')  break;
        scanf("%s", pattern);                        //读模式串
        cnt = 0;
```

```
        kmp(str, pattern);
        printf("%d\n", cnt);
    }
    return 0;
}
```

【习题】

hdu 1686/1711/2222/2896/3065/3336。

hdu 2594"Simpsons' Hidden Talents",扩展 KMP 算法,求原串 S 的每一个后缀子串与模式串 P 的最长公共前缀。

9.5 AC 自动机

AC 自动机(Aho-Corasick automaton)是 KMP 的升级版。KMP 是单模匹配算法,处理在一个文本串中查找一个模式串的问题;AC 自动机是多模匹配算法,能在一个文本串中同时查找多个不同的模式串。

多模匹配问题:给定一个长度为 n 的文本 S,以及 k 个平均长度为 m 的模式串 $P_1,P_2,\cdots,P_k$,要求搜索这些模式串出现的位置。

其实用 KMP 也能解决多模匹配问题,缺点是复杂度较高,需要对每个 $P_1,P_2,\cdots,P_k$ 分别做一次 KMP,总复杂度是 $O((n+m)k)$。

AC 自动机算法并不需要对 S 做多次 KMP,而是只搜索一遍 S,在搜索时匹配所有的模式串。

如何同时匹配所有的 P? 如果读者能结合前面介绍过的字典树就恍然大悟了。

KMP 是通过查找 P 对应的 Next[]数组实现快速匹配的。如果把所有的 P 做成一个字典树,然后在匹配的时候查找这个 P 对应的 Next[]数组,不就实现了快速匹配的效果吗?

复杂度分析:k 个模式串,平均长度为 m;文本串长度为 n。建立字典树 $O(km)$;建立 fail 指针 $O(km)$;模式匹配 $O(nm)$,乘 m 的原因是在统计的时候需要顺着链回溯到 root 结点。总时间复杂度是 $O(km+km+nm)=O(km+nm)$。

对比简单使用 KMP 的复杂度 $O((n+m)k)$,当 $m \ll k$ 时,$(k+n)m \ll (n+m)k$。AC 自动机优势非常大。

hdu 2222 题是一道模板题。

hdu 2222 "Keywords Search"

有多个关键词,在一个文本中找到它们。

输入:第 1 行是测试用例个数。每个用例包括一个整数 N,表示关键词个数,下面有 N 个关键词,$N \leqslant 10\,000$。每个关键词只包括小写字母,长度不超过 50。最后一行是文本,长度不大于 1 000 000。

输出:在输出文本中能找到多少关键词。重复的关键词只需要统计一次。

下面是代码[①]。

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 1000000 + 100;
const int SIGMA_SIZE = 26;
const int maxnode = 1000000 + 100;
int n, ans;
bool vis[maxn];
map< string, int > ms;
int ch[maxnode][SIGMA_SIZE + 5];
int val[maxnode];
int idx(char c) {return c - 'a';}
struct Trie {
    int sz;
    Trie() { sz = 1; memset(ch[0], 0, sizeof(ch[0])); memset(vis, 0, sizeof(vis)); }
    void insert(char * s) {
        int u = 0, n = strlen(s);
        for(int i = 0; i < n; i++) {
            int c = idx(s[i]);
            if(!ch[u][c]) {
                memset(ch[sz], 0, sizeof(ch[sz]));
                val[sz] = 0;
                ch[u][c] = sz++;
            }
            u = ch[u][c];
        }
        val[u]++;
    }
};
//AC 自动机
int last[maxn], f[maxn];
void print(int j) {
 if(j && !vis[j]) {
    ans += val[j]; vis[j] = 1;
    print(last[j]);
 }
}
int getFail() {
  queue< int > q;
  f[0] = 0;
  for(int c = 0; c < SIGMA_SIZE; c++) {
      int u = ch[0][c];
      if(u) {f[u] = 0; q.push(u); last[u] = 0;}
  }
  while(!q.empty()) {
      int r = q.front(); q.pop();
      for(int c = 0; c < SIGMA_SIZE; c++) {
          int u = ch[r][c];
          if(!u) {
              ch[r][c] = ch[f[r]][c];
```

① 其中，getFail()来自《算法竞赛入门经典训练指南》，刘汝佳，陈锋，清华大学出版社，3.3.3 节，214 页。

```
                continue;
            }
            q.push(u);
            int v = f[r];
            while(v && !ch[v][c]) v = f[v];
            f[u] = ch[v][c];
            last[u] = val[f[u]] ? f[u] : last[f[u]];
        }
    }
}
void find_T(char * T) {
  int n = strlen(T);
  int j = 0;
  for(int i = 0; i < n; i++) {
      int c = idx(T[i]);
      j = ch[j][c];
      if(val[j]) print(j);
      else if(last[j]) print(last[j]);
  }
}
char tmp[105];
char text[1000000 + 1000];
int main() {
  int T; cin >> T;
  while(T--) {
      scanf("%d", &n);
      Trie trie;
      ans = 0;
      for(int i = 0; i < n; i++) {
          scanf("%s", tmp);
          trie.insert(tmp);
      }
      getFail();
      scanf("%s", text);
      find_T(text);
      cout << ans << endl;
    }
      return 0;
}
```

【习题】

hdu 2243/2825/2296，AC 自动机＋DP 状态压缩。

9.6　后缀树和后缀数组

后缀树和后缀数组理解起来比较难，但是可以解决大部分字符串问题，前面提到的字符串匹配问题，例如查找子串、最长重复子串、最长公共子串等，都可以用后缀数组解决，这类题目是编程竞赛的常见题型。

本节首先讲解后缀树和后缀数组的概念，然后用后缀数组解决一些经典字符串问题。

9.6.1 概念

后缀(suffix)：一个字符串，它的一个后缀是指从某个位置开始到末尾的一个子串。例如字符串 string s＝"vamamadn"，它的后缀有 8 个，即 s[0]＝"vamamadn"、s[1]＝"amamadn"、s[2]＝"mamadn"等。具体见表 9.1 的左半部分。

表 9.1 后缀

后缀 $s[i]$	下标 i		字典序	后缀数组 sa[j]	下标 j
vamamadn	0		adn	5	0
amamadn	1		amadn	3	1
mamadn	2		amamadn	1	2
amadn	3		dn	6	3
madn	4		madn	4	4
adn	5		mamadn	2	5
dn	6		n	7	6
n	7		vamamadn	0	7

后缀树(suffix tree)：就是把所有的后缀子串用字典树的方法建立的一棵树，如图 9.5 所示。

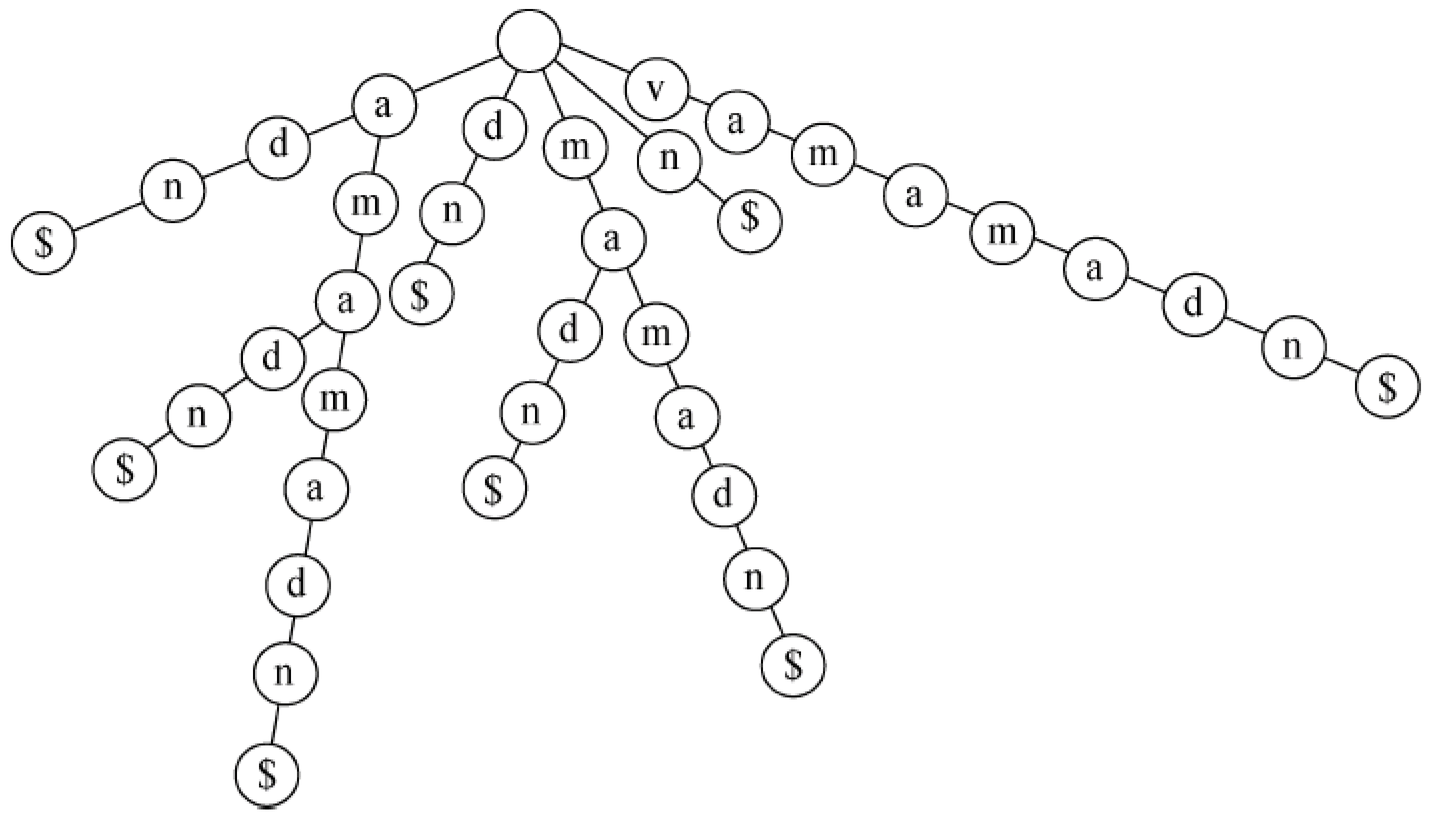


图 9.5 后缀树

其中，根结点为空，符号 $ 表示一个后缀子串的末尾。用 $ 的原因是它比较特殊，不会在字符串中出现，适合用来做标识。如果要利用后缀树查找某个子串，例如"mam"，只需要从根结点出发查 3 次即可，这就是后缀树的优势。

由于直接对后缀树进行构造和编程不太方便，所以用后缀数组(suffix array)这种简单的方法来替代。在表 9.1 中，后缀数组就是按字典序对应的后缀下标：int sa[]＝{5,3,1,6,4,2,7,0}。很明显，后缀数组的数字顺序就是后缀子串的字典顺序，记录了子串的有序排列。例如 sa[0]＝5，意思是：排名 0(即字典序最小)的子串，是原字符串中从第 5 个位置开始的后缀子串，即"adn"。

如果得到了后缀数组，可以很方便地解决一些字符串问题。下面介绍查找子串（单模匹配）问题，即在母串 s 中查找子串 t。只需要在后缀数组 sa[]上做二分搜索，就能很快地找到子串。比如查找子串 t="ad"，程序如下：

```
#include <bits/stdc++.h>
using namespace std;
int find(string s, string t, int * sa){        //在 s 中查找子串 t; sa 是 s 的后缀数组
    int i = 0, j = s.length();
    while(j - i > 1) {
        int k = (i + j)/2;                     //二分法，操作 O(logn)次
        if(s.compare(sa[k], t.length(), t)< 0) //匹配一次，复杂度是 O(m)
            i = k;
        else j = k;
    }
    if(s.compare(sa[j], t.length(), t) == 0)   //找到了，返回 t 在 s 中的位置
        return sa[j];
    if(s.compare(sa[i], t.length(), t) == 0)
        return sa[i];
    return -1;                                 //没找到
}
int main(){
    string s = "vamamadn", t = "ad";           //母串和子串
    int sa[] = {5, 3, 1, 6, 4, 2, 7, 0};       //sa[]是 s 的后缀数组，假设已经得到了
    int location = find(s, t, sa);
    cout << location <<":"<< &s[location]<< endl << endl; //打印 t 在 s 中的位置
}
```

每次查找，复杂度都是 $O(m\log_2 n)$，m 是子串长度，n 是母串长度。

在上面的程序里事先已经算好了后缀数组 sa[]，所以最关键的问题是如何高效地求后缀数组？即如何对后缀子串进行排序？

常用的一种排序方法为倍增法，它的复杂度是 $O(n\log_2 n)$，下一节将详细介绍这个方法。

后缀数组是很高效的方法。例如在上面的查找子串问题中先求后缀数组，再找子串，总复杂度是 $O(n\log_2 n+m\log_2 n)$。对比经典的字符串匹配 KMP 算法，复杂度是 $O(n+m)$，前缀数组已经很接近了。如果直接用后缀树，速度更快：建树的复杂度是 $O(mn)$，在树上查找一个子串只需要比较 m 次，复杂度是 $O(m)$。

对比后缀数组和后缀树，根据前面的讲解可以知道，后缀树用空间换时间，复杂度很好；后缀数组虽然复杂度稍微差一点，但是使用的空间小，编码简单，所以在竞赛中一般使用后缀数组。

9.6.2 用倍增法求后缀数组

在讲解倍增法之前先考虑常见的排序方法，例如快速排序。快速排序，所有元素的比较次数是 $O(n\log_2 n)$，在应用到字符串排序时，每两个字符串还有 $O(n)$的比较，所以总复杂度是 $O(n^2\log_2 n)$，显然不够好。

用倍增法对后缀排序的原理比较复杂,初学者很难理解,不过如果读者按以下步骤学习就会觉得很清晰。

例:求字符串"vamamadn"的后缀数组。

第1步:用数字代表字母,例如a最小,记为0;v最大,记为4。这个转换对后缀子串的排序没有影响(这一步操作实际上是对所有的后缀子串的最高位进行大小判定,不过因为很多子串的最高位相同,对应的数字也相同,所以还不能比较大小)。

第2步:连续两个数字的组合,相当于连续两个字符。例如40代表"va"、02代表"am"等。最后一个3没有后续,在尾部加上0,组成30。这并不影响字符的比较,因为字符是从头到尾比较大小的(这一步操作是取后缀子串的最高两位,数字的大小代表子串的最高两位的大小)。

第3步:连续4个数字的组合,相当于连续4个字符。例如4020代表"vama"、0202代表"amam"等。最后的30没有后续,加上00,组成3000(这一步操作是用数字代表后缀子串的高4位)。

特别需要注意的是,并没有进行连续3个数字的组合。原因有两个,一是不方便操作,二是并不影响后缀子串的大小比较。

在第3步操作后产生的8个数字已经全部不一样,能区分大小了。结束,并进行排序,得到rk[]={7,2,5,1,4,0,3,6}。rk是rank的缩写,表示"名次数组"。rk[]是字符串"vamamadn"的8个后缀子串的排序。在得到rk[]后,可以求得后缀数组sa[]={5,3,1,6,4,2,7,0},如图9.6所示。

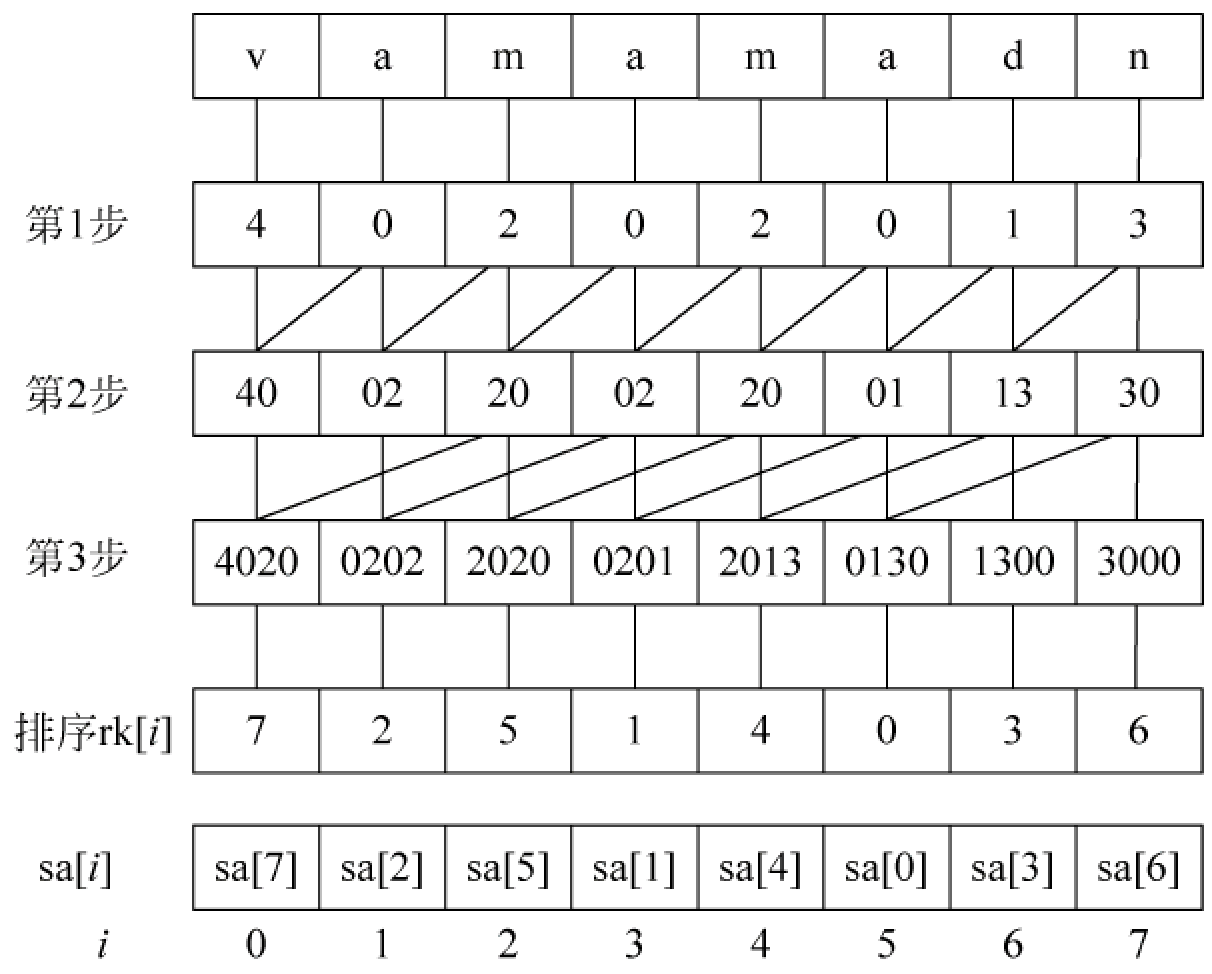


图9.6　名次数组和后缀数组

上述操作,因为每一步都递增两倍,所以总步骤一共有log(n)步,非常少。

虽然上述过程看起来很不错,但是却并不实用。因为字符串可能很长,例如包含1万个字符,那么在最后一步产生的每个数字都有10 000位,是个天文数字,根本无法存储和排序。

那么能不能在每一步中缩小产生的组合数字的大小,而且还能保持顺序呢?答案是能。方法是在每一步操作后就对组合数字进行排序,用序号产生一个新数字,然后用新数字进

行下一步操作，过程如图 9.7 所示。

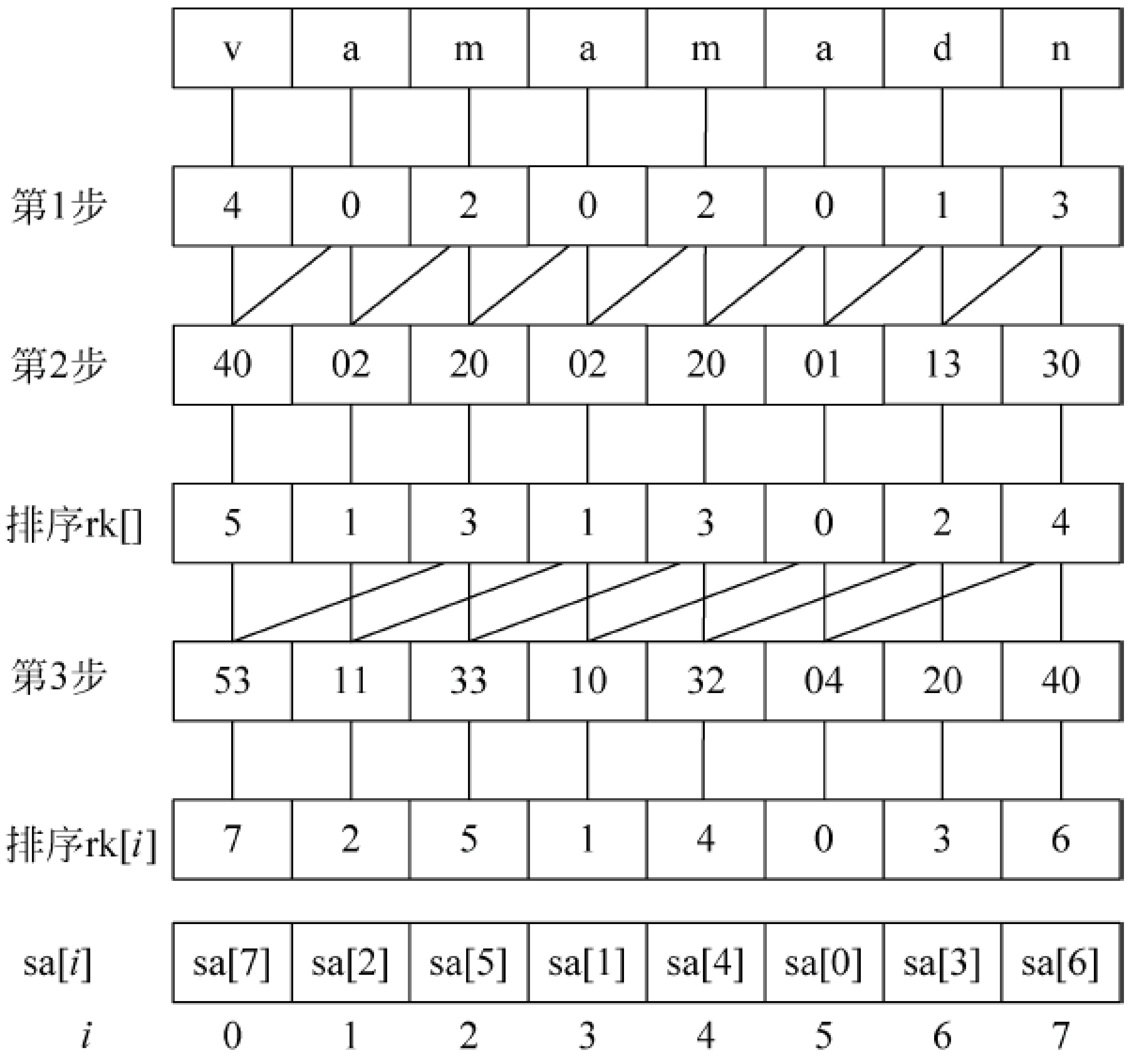


图 9.7　改进后的名次数组和后缀数组

可以发现，每一步排序后产生的新数字实际上仍然是对后缀子串的高位的排序。所以，最后的结果和图 9.6 是一样的。

产生的新数字有多大？假设字符串长度 $n=1$ 万，即每一步处理 1 万个数，那么产生的新数字是对这 1 万个数的排序结果，最大就是 10 000。所以，每一步的排序只是对 1 万个大小在 1～10 000 的数字进行排序，这是很容易做到的。

在这个过程中，核心是处理 rk[]和 sa[]。

1. sa[]、rk[]数组

在后缀数组的相关程序中，有 3 个关键的数组：sa[]、rk[]和 height[]。下面给出 sa[]、rk[]的概念和相互关系，请对照图 9.7 进行理解。

sa[]：后缀数组 suffix array。保存 $0\sim n-1$ 的全排列，含义是，把所有后缀按字典序排序后，后缀在原串中的位置。性质：suffix(sa[i])< suffix(sa[$i+1$])。sa[]记录“位置”：“排第 i 的是谁?”——“排第 i 的后缀子串在原串的 sa[i]这个位置。”

rk[]：名次数组 rank array。最后得到的 rk[]也是 $0\sim n-1$ 的全排列，保存 suffix(i)在所有后缀中按字典序排序的“名次”。rk[]记录“排名”：第 i 个后缀子串排第几?”——“原串从头数第 i 个后缀子串，排名是 rk[i]。”

rk[]和 sa[]是一一对应关系，互为逆运算，可以互相推导：

(1) 用 rk[]推导 sa[]：

```
for(int i = 0; i < n; i++)    sa[rk[i]] = i;
```

(2) 用 sa[]推导 rk[]：

```
for(int i = 0; i < n; i++)  rk[sa[i]] = i;
```

2. 用 sort()函数求后缀数组 sa[]

下面用 STL 的 sort()函数对 rk[]排序,并求得后缀数组。程序的核心就是上面两个推导,读者需要透彻理解才能看懂下面的代码。

比较函数 comp_sa()判断每一步中得到的组合数的大小。在图 9.7 所示的原理图中,例如从第 1 步到第 2 步,把 4、0 组合成 40,把 0、2 组合成 02,等等,然后再用于比较。comp_sa()省去了组合过程,直接进行比较：首先比较 40 和 02 的高位,再比较低位。

程序的逻辑如下：

(1) 用 sort()在每一步根据当前的 rk[]计算出当前的 sa[],请读者认真体会细节。

(2) 用 sa[]更新下一步用到的 rk[]。注意每一步的 sa[],其中任意两个 sa[i]和 sa[j]都不同,但是下一步的 rk[]中有一些是相同的,所以 sa[]和 rk[]还不是一一对应的。此时需要先用 sa[]根据原来的 rk[]中的记录推导新的 rk[],这需要用一个临时 tmp[]存放新值,然后再赋值给 rk[]。只有到了最后,sa[]和 rk[]才是一一对应的。

计算后缀数组 sa[]的模板[①]

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 200005;                //字符串的长度
char s[MAXN];                           //输入字符串
int sa[MAXN], rk[MAXN], tmp[MAXN + 1];
int n, k;

bool comp_sa(int i, int j){             //组合数有两个部分,高位是 rk[i],低位是 rk[i+k]
    if(rk[i] != rk[j])                  //先比较高位: rk[i]和 rk[j]
        return rk[i] < rk[j];
    else{                               //高位相等,再比较低位的 rk[i+k]和 rk[j+k]
        int ri = i+k<=n? rk[i+k] : -1;
        int rj = j+k<=n? rk[j+k] : -1;
        return ri < rj;
    }
}
void calc_sa() {                        //计算字符串 s 的后缀数组
    for(int i = 0; i<=n; i++)    {
        rk[i] = s[i];                   //字符串的原始数值
        sa[i] = i;                      //后缀数组,在每一步记录当前排序后的结果
    }
    for(k=1; k<=n; k = k*2){            //开始一步步操作,每一步递增两倍进行组合
        sort(sa, sa+n, comp_sa);        //排序,结果记录在 sa[]中
        tmp[sa[0]] = 0;
        for(int i = 0; i<n; i++)        //用 sa[]倒推组合数,并记录在 tmp[]中
          tmp[sa[i+1]] = tmp[sa[i]] + (comp_sa(sa[i],sa[i+1]) ? 1: 0);
        for(int i = 0; i<n; i++)        //把 tmp[]复制给 rk[],用于下一步操作
          rk[i] = tmp[i];
    }
```

① 参考《挑战程序设计竞赛》,秋叶拓哉,379 页,“4.7.3 后缀数组”。

```
}
int main(){
    while(scanf("%s",s)!= EOF){          //读字符串
        n = strlen(s);
        calc_sa();                        //求后缀数组 sa[]
        for(int i = 0;i < n;i++)          //打印后缀数组
            cout << sa[i]<<" ";
    }
    return 0;
}
```

上面的程序用到的 sort()实际是快速排序，每一步排序的复杂度是 $O(n\log_2 n)$，一共有 $\log_2 n$ 个步骤，总复杂度是 $O(n\log_2 n)$。虽然已经很好了，不过还有一种更快的排序方法——基数排序，总复杂度只有 $O(n\log_2 n)$。在下一节的问题 hdu 1403 中分别提交用 sort()和基数排序两种方案的倍增法程序，执行时间分别是 1000ms 和 80ms。

3. 基数排序

基数排序是一种不太符合常识的排序方法，它不是先比较高位再比较低位，而是反过来，先比较低位再比较高位。

例如排序{47,23,19,17,31}：

第 1 步：先按个位大小排序，得到{31,23,47,17,19}；

第 2 步：再按十位大小排序，得到{17,19,23,31,47}，结束，得到有序排列。

更特别的是，上述操作并不是用比较的方法得到的，而是用"哈希"的思路：直接把数字放到对应的"格子"里，第 1 步按个位放，第 2 步按十位放。表 9.2 中第 2 步得到的序列就是结果。

表 9.2　把数字放到对应的"格子"里

格　　子	0	1	2	3	4	5	6	7	8	9
第 1 步		**31**		**23**				**47**、**17**		**19**
第 2 步		**17**、**19**	**23**	**31**	**47**					

基数排序的复杂度：n 个数，每个数有 d 位（例如上面例子中的 17～47 都是两位数），每一位有 k 种可能（十进制，0～9 共 10 种情况），复杂度是 $O(d(n+k))$，存储空间是 $O(n+k)$。对长度 10 000 的字符串进行一次排序，$n=10\ 000$，$d\leqslant 5$，$k=10$，复杂度 $d(n+k)\leqslant 10\ 000\times 5$，而一次快排的复杂度 $n\log_2 n\approx 10\ 000\times 13$。

对比快速排序等排序方法，基数排序在 d 比较小的情况下（即所有的数字差不多大时）是更好的方法。如果 d 比较大，基数排序并不比快速排序更好。

下面的程序用基数排序求后缀数组①。

```
//程序的 main()部分和上面用 sort()时的一样
char s[MAXN];
int sa[MAXN],cnt[MAXN],t1[MAXN],t2[MAXN],rk[MAXN],height[MAXN];
```

① 代码改编自《算法竞赛入门经典训练指南》，刘汝佳，陈锋，清华大学出版社，3.4.1 节。

```
int n;
void calc_sa() {      //void build_sa(int n, int m)     //n 是字符串长度
    int m = 127;                               //m 是小写字母的 ASCII 码值范围.构造字符串 s 的后
                                               //缀数组,每个字符的值必须为 0~m-1
    int i, *x = t1, *y = t2;
    for(i = 0;i < m;i++)   cnt[i] = 0;
    for(i = 0;i < n;i++)   cnt[x[i] = s[i]]++;
    for(i = 1;i < m;i++)   cnt[i] += cnt[i-1];
    for(i = n-1;i>= 0;i-- )  sa[ --cnt[x[i]]] = i;
    //sa[]: 从 0 到 n-1
    for(int k = 1;k <= n;k = k * 2){           //利用对长度为 k 的排序的结果对长度为 2k 的排序
        int p = 0;
        //2nd
        for(i = n-k;i < n;i++)  y[p++] = i;
        for(i = 0;i < n;i++)    if(sa[i]>= k) y[p++] = sa[i] - k;
        //1st
        for(i = 0;i < m;i++)   cnt[i] = 0;
        for(i = 0;i < n;i++)   cnt[x[y[i]]]++;
        for(i = 1;i < m;i++)   cnt[i] += cnt[i-1];
        for(i = n-1;i>= 0;i-- )  sa[ --cnt[x[y[i]]]] = y[i];
        swap(x,y);
        p = 1; x[sa[0]] = 0;
        for(i = 1;i < n;i++)
            x[sa[i]] =
              y[sa[i-1]] == y[sa[i]]&&y[sa[i-1] + k] == y[sa[i] + k]?p-1:p++;
        if(p >= n) break;
        m = p;
    }
}
```

4. 高度数组 height[]

height[]是一个辅助数组,和最长公共前缀(Longest Common Prefix,LCP)相关。height[]数组非常重要,很多使用后缀数组解决的题目都依赖 height[]数组完成。

LCP(i,j):suffix(sa[i])与 suffix[sa[j]]的最长公共前缀长度,即排序后第 i 个后缀和第 j 个后缀的最长公共前缀长度。

LCP(i,j)=min{LCP($k-1$,k)},$i<k\leqslant j$

定义 height[i]为 sa[$i-1$]和 sa[i],也就是排名相邻的两个后缀的最长公共前缀长度。例如前面的"vamamadn"中,sa[1]表示"amadn",sa[2]表示"amamadn",那么 height[2]=3,表示 sa[1]和 sa[2]这两个后缀的前 3 个字符相同。

用暴力的方法可以推导 height[]数组,即比较所有相邻的 sa[],然而复杂度是 $O(n^2)$。下面给出复杂度为 $O(n)$的代码:

```
void getheight(int n){                      //n 是字符串长度
    int i, j, k = 0;
    for(i = 0 ;i < n; i++)  rk[sa[i]] = i;  //用 sa[]推导 rk[]
    for(i = 0; i < n; i++) {
        if(k)  k-- ;
```

```
        int j = sa[rk[i] - 1];
        while(s[i + k] == s[j + k])  k++;
        height[rk[i]] = k;
    }
}
```

height[]数组的应用非常广泛,其中最直接的应用是求最长重复子串问题、求最长公共子串问题,见下一节的讨论。

9.6.3 用后缀数组解决经典问题

在字符串问题中有这样一些经典问题,可以用后缀数组解决:

(1) 在字符串 s 中查找子串 t,具体操作见 9.6.1 节。

(2) 在字符串 s 中找最长重复子串。先求 height[]数组,其中的最大值 height[i]就是最长重复子串的长度。如果需要打印最长重复子串,它就是后缀子串 sa[$i-1$]和 sa[i]的最长公共前缀。

(3) 找字符串 s_1 和 s_2 的最长公共子串,以及扩展到求多个字符串的最长公共子串。最长公共子串(Longest Common Substring)和最长公共子序列(Longest Common Subsequence)不同。子串是串的一个连续的部分,子序列则不必连续。比如字符串"abcf"和"bcef"的最长公共子串为"bc",而最长公共子序列是"bcf"。这两个问题,在数据规模小的情况下都可以用动态规划求解,设 s_1、s_2 的长度分别是 m、n,则复杂度是 $O(mn)$。然而动态规划并不够好,如果 $m,n>10\ 000$,动态规划就不能用了,需要用后缀数组。

这个问题实际上和上一个问题"最长重复子串"类似:合并 s_1 和 s_2,得到一个大字符串 s,就变成了上一个问题。技巧是在合并的时候需要在 s_1 和 s_2 之间插入一个未出现过的特殊字符,例如'$',进行分隔,避免合并产生更长的子串。

具体操作:首先计算 height[]数组,然后查找最大的 height[i],而且它对应的 sa[$i-1$]和 sa[i]分别属于被'$'分隔的前后两个字符串时,就是解。

hdu 1403 题是最长公共子串问题。

hdu 1403 "最长公共子串"

求两个字符串的最长公共子串。

输入:每个测试用例包含两个字符串,每个字符串最多有 100 000 个字符。所有的字符都是小写的。

输出:输出最长公共子串的长度。

输入样例:

banana

cianaic

输出样例:

3

在样例中,最长公共子串是"ana",长度是 3。由于字符串长度是 100 000,程序的复杂

度不能大于 $O(n\log_2 n)$。

下面给出用后缀数组实现的程序。其中用到的 calc_sa()、getheight()函数已经在前文给出。读者可以分别用 sort()和基数排序实现的 calc_sa()提交，经验证，sort()版程序的执行时间是 1000ms，基数排序版的是 80ms。

最长公共子串程序

```
//省略了 calc_sa()、getheight()函数，已在上一节给出
int main(){
    int len1, ans;
    while(scanf("%s", s)!= EOF) {          //读第 1 个字符串
        n = strlen(s);
        len1 = n;
        s[n] = '$';                         //用'$'分隔两个字符串
        scanf("%s", s+n+1);                 //读第 2 个字符串，与第 1 个合并
        n = strlen(s);
        calc_sa();                          //求后缀数组 sa[]
        getheight(n);                       //求 height[]数组
        ans = 0;
        for(int i = 1; i<n; i++)
         //找最大的 height[i]，并且它对应的 sa[i-1]和 sa[i]分别属于前后两个字符串
           if(height[i]>ans &&
           ((sa[i-1]<len1 &&sa[i]>=len1) || (sa[i-1]>=len1&&sa[i]<len1)))
               ans = height[i];
        printf("%d\n",ans);
    }
    return 0;
}
```

(4) 找字符串 s 的最长回文子串。例如"helpsoshelp"的最长回文子串是"sos"。回文串一般用 Manacher 算法。

【习题】

hdu 5769，后缀数组。

hdu 3948，回文串。

hdu 4691，最长公共前缀。

hdu 5008，第 k 小子串。

hdu 4416，后缀自动机。

9.7 小　　结

本章讲解的 KMP、AC 自动机、后缀数组等知识点都比较复杂，并且在应用中经常结合 DP 等其他算法。字符串算法也是算法竞赛中的难点。

第 10 章　图　　论

- ✍图的概念和存储
- ✍图的遍历和连通性
- ✍拓扑排序
- ✍欧拉路
- ✍无向图和有向图的连通性
- ✍2-SAT 问题
- ✍最短路径
- ✍最小生成树
- ✍最大流：残留网络、增广路
- ✍最小割
- ✍最小费用最大流
- ✍二分图匹配

图是一种很常见的模型，能描述事物或状态的关系，很多问题可以抽象为图论问题。图论的算法十分丰富，常见的问题或算法有 60 多个。在算法竞赛中，图论属于比较难的内容。

本章讲解图论的基本概念、图论常用的数据结构、常见的图论、网络流算法，并通过经典题目分析建模过程，给出标准程序。

10.1　图的基本概念

图是常见的抽象模型，由点(node，或者 vertex)和连接点的边(edge)组成。图是点和边构成的网。图描述了事物之间的连接。图最典型的应用场景是地图，地图由地点和道路组成，它的特征如下。

(1) 地点：可能是十字路口，也可能是三岔路口，或者仅仅是一个连接点。在图论中，把地点抽象为点。

(2) 道路：可能是单行道或双行道。抽象成有向边或无向边。

(3) 道路有过路费：抽象成边的权值。

(4) 求两点间的最短道路，即图论里的最短路径算法。

(5) 在城市群之间如何修最短的连通道路，即图论中的最小生成树问题。

地图的这些问题都是图论研究的对象。

计算机网络也是典型的图问题，和地图非常相似。

人际关系也可以抽象成图，即社交网络。例如著名的“六度空间理论”，世界上任意两个人，最多通过 5 个中间人就能联系到。把人看成点，把人和人之间的关系看成边，这就是一

个图的连通性问题。

树，即连通无环图，它是一种特殊的图。树的结点从根开始，层层扩展子树，是一种层次关系，这种层次关系保证了树上的结点不会出现环路。在图的算法中，经常需要在图上生成一棵树，再进行操作。

根据边有无方向、有无权值、有无环路，可以把图分成很多种，例如：

(1) 无向无权图，边没有权值、没有方向；

(2) 有向无权图，边有方向、无权值；

(3) 加权无向图，边有权值，但没有方向；

(4) 加权有向图；

(5) 有向无环图(Directed Acyclic Graph，DAG)。

图算法的复杂度显然和边的数量 E、点的数量 V 相关。如果一个算法的复杂度是线性时间 $O(V+E)$，这几乎是图问题中能达到的最好程度了。如果能达到 $O(V\log_2 E)$、$O(E\log_2 V)$或类似的复杂度，则是很好的算法。如果是 $O(V^2)$、$O(E^2)$或更高，在图问题中不算是好的算法。

10.2 图的存储

对图的任何操作都需要基于一个存储好的图。图的存储结构必须是一种有序的存储，能让程序很快定位结点 u 和 v 的边(u,v)，最好能在 $O(1)$的时间内只用一次或几次就定位到。

一般用 3 种数据结构存储图，即邻接矩阵、邻接表、链式前向星。

以图 10.1 所示的有向图为例，图中有 6 个结点、11 条边。

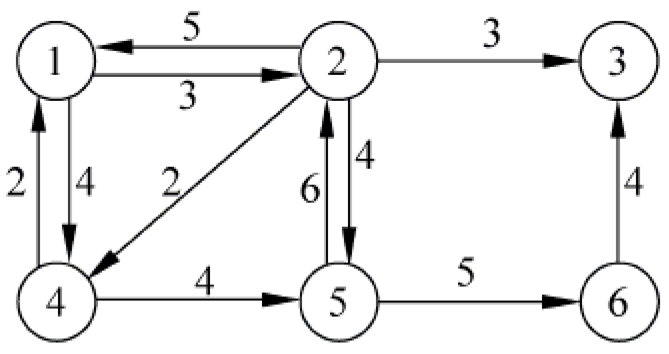


图 10.1 有向图

1. 邻接矩阵

用二维数组存储即可：int graph[NUM][NUM]。

无向图：graph[i][j]=graph[j][i]。

有向图：graph[i][j]!=graph[j][i]。

权值：graph[i][j]存结点 i 到 j 的边的权值，例如 graph[1][2]=3、graph[2][1]=5 等。用 graph[i][j]=INF 表示 i 和 j 之间无边。

优点：适合稠密图；编码非常简短；对边的存储、查询、更新等操作又快又简单，只需要一步就能访问和修改。

缺点：

(1) 存储复杂度 $O(V^2)$太高。如果用来存稀疏图，大量空间会被浪费。例如上面的图，6 个点，只有 11 条边，但是 graph[6][6]的空间是 36。当 $V=10\ 000$ 个结点时，空间为 100MB，已经超过了常见 ACM 竞赛题的空间限制，而一百万个点的图在 ACM 题中是很常见的。

(2) 一般情况下不能存储重边。(u,v)之间可能有两条或更多的边，这些边的费用不同、容量不同，是不能合并的。有向边(u,v)在矩阵中只能存储一个参数，矩阵本身的局限

性使它不能存储重边。不过，如果这个参数值只是用来表示边的数量，也算是存储了重边。

2. 邻接表

邻接表的概念请阅读《数据结构》教材，规模大的稀疏图一般用邻接表存储。它的优点是存储效率非常高，只需要与边数成正比的空间，存储复杂度为 $O(V+E)$，几乎已经达到了最优的复杂度，而且能存储重边；缺点是编程比邻接矩阵麻烦一些，访问和修改也慢一些。

在本章“10.9.3 SPFA”这一节中用 STL 的 vector 实现了邻接表，有关代码如下：

```
//定义边
    struct edge{
        int from, to, w;                                  //边：起点 from，终点 to，权值 w
        edge(int a, int b,int c){from = a; to = b; w = c;} //对边赋值
    };
    vector < edge > e[NUM];                               //e[i]：存第 i 个结点连接的所有边
//初始化
    for(int i = 1; i <= n; i++)
        e[i].clear();
//存边
     e[a].push_back(edge(a,b,c));                         //把边(a,b)存到结点 a 的邻接表中
//检索结点 u 的所有邻居
     for(int i = 0; i < e[u].size(); i++){                //结点 u 的邻居有 e[u].size()个
        …
     }
```

例如，在上面的图 10.1 中，结点 2 的邻接表是(2 1 5)→(2 3 3)→(2 4 2)→(2 5 4)。

3. 链式前向星

用邻接表存图非常节省空间，一般的大图也够用了。然而，如果空间极其紧张，有没有更紧凑的存图方法呢？邻接表有没有改进的空间？

分析邻接表的组成，存储一个结点 u 的邻接边，其方法的关键是先定位第 1 个边，第 1 个边再指向第 2 个边，第 2 个边再指向第 3 个边，依此类推，根据这个分析，可以设计一种极为紧凑、没有任何空间浪费、编码非常简单的存图方法。图 10.2 是对前面的图 10.1 生成的存储空间，其中，head[NUM]是一个静态数组，struct edge 是一个结构的静态数组。

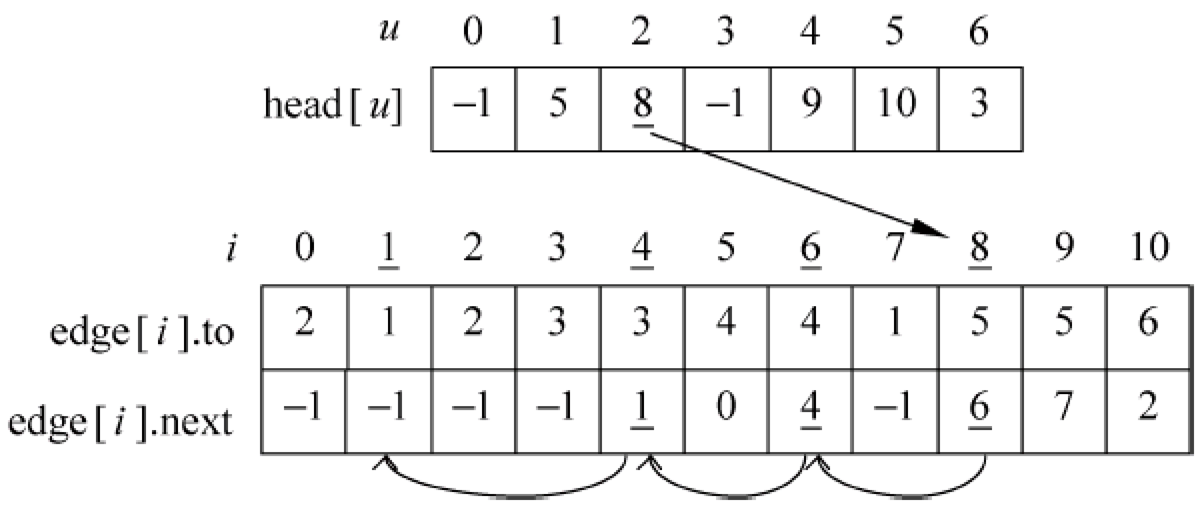


图 10.2 链式前向星存图

以结点 2 为例，从点 2 出发的边有 4 条，即(2,1)、(2,3)、(2,4)、(2,5)，邻居是 1、3、4、5。

(1) 定位第 1 个边。用 head[]数组实现，例如 head[2]指向结点 2 的第 1 个边，head[2]=8，它存储在 edge[8]这个位置。

(2) 定位其他边。用 struct edge 的 next 参数指向下一个边。edge[8]. next＝6，指向下一个边在 edge[6]这个位置，然后 edge[6]. next＝4，edge[4]. next＝1，最后 edge[1]. next＝－1，－1 表示结束。

struct edge 的 to 参数记录这个边的邻居结点。例如 edge[8]. to＝5，第一个邻居是点 5；然后 edge[6]. to＝4，edge[4]. to＝3，edge[1]. to＝1，得到邻居是 1、3、4、5。

上述存储方法被称为**"链式前向星"**，它是空间效率最高的存储方法，因为它用静态数组模拟邻接表，没有任何浪费。

那么如何生成上述的存储结果？下面的程序片段来自后面 SPFA 这一节的例子。每执行一次 addedge()，就把一个新的边存入空间。

按以下顺序处理图中所有的边(u，v)：(1，2)、(2，1)、(5，2)、(6，3)、(2，3)、(1，4)、(2，4)、(4，1)、(2，5)、(4，5)、(5，6)，得到图 10.2。输入的顺序会影响结果。

从执行过程可知，每加入一个新的边，都是直接加在整个 edge[]的末尾，而与这个边的特征毫无关系。

下面是程序。

```
const int NUM = 1000005;            //一百万个点，一百万个边
struct Edge{
  int to, next, w;                  //边：终点 to、权值 w、下一个边 next.起点放在 head[]中
}edge[NUM];
int head[NUM];                      //head[u]指向结点 u 的第一个边的存储位置
int cnt;                            //记录 edge[]的末尾位置，新加入的边放在末尾
void init(){                        //初始化
   for(int i = 0; i < NUM; ++i){
      edge[i].next = -1;            //-1：结束，没有下一个边
      head[i] = -1;                 //-1：不存在从结点 i 出发的边
   }
   cnt = 0;
}
void addedge(int u, int v, int w){
  edge[cnt].to = v;
  edge[cnt].w = w;
  edge[cnt].next = head[u];         //指向结点 u 上一次存的边的位置
  head[u] = cnt++;                  //更新结点 u 最新边的存放位置：就是 edge 的末尾
}
//遍历结点 i 的所有邻居
for(int i = head[u]; ~i; i = edge[i].next)  //~i 也可以写成 i!= -1
{   …   }
```

链式前向星的优点是存储效率高、程序简单、能存储重边；缺点是不方便做删除操作。作为练习，读者可以自己编写删除的程序。

链式前向星的例程见 10.9 节。

10.3 图的遍历和连通性

图的基本特征是点和边，图的基本算法是用搜索来处理点和边的关系。第 4 章介绍了用 BFS 和 DFS 遍历一个图，在遍历的同时也解决了图的连通性问题。BFS 和 DFS 是图论

的基本算法，本章大部分内容是基于它们的。这些算法，或者直接用 BFS 和 DFS 来解决问题，或者用其思想建立新的算法。请读者回顾 BFS 和 DFS 的内容，透彻理解并能熟练写出程序。

特别是 DFS，用递归来搜索图，比 BFS 更难理解；但是一旦理解之后，编程将十分便利。图论中的很多算法，例如拓扑排序、强连通分量等，都建立在 DFS 之上。

下面是 DFS 的示例程序，其中用 vector 邻接表来存图。用矩阵存图的 DFS 示例见第 4 章。

```
vector<int> G[N];                         //G[u][i]：第 u 个结点直连的第 i 个结点
int vis[N];                               //点的访问标志，vis = 0 表示未访问过
                                          //vis = 1 表示已经被正常处理过
                                          //vis = -1 表示正在被访问中，这在有些判断中有用
                                          //例如在拓扑排序中，用于判断跳出死循环
bool dfs(int u) {                         //以 u 为起点开始 DFS 搜索
    vis[u] = 1;                           //在本次递归中被正常访问
    { … ; return true;}                   //出现目标状态，正确返回
    { … ; return false;}                  //做相应处理，返回错误
    for(int i = 0; i < G[u].size(); i++ ) { //u 的邻居有 G[u].size()个
       int v = G[u][i];                   //v 是第 i 个邻居
       if(!vis[v])                        //如果 v 没有访问过
           return dfs(v);                 //递归访问第 v 个邻居
    }
    { … ; }                               //事后处理，返回正确或错误
}
```

下面用图 10.3 所示的例子来帮助读者理解 DFS 在图中的应用。这个例子故意设计成非连通图，所以从一个点出发并不能访问到所有点。

1. 求某个点的连通性

对需要的点执行 dfs()，就能找到它连通的点。例如找图 10.3 中 e 点的连通性，执行 dfs(e)，访问过程见图 10.4 结点上面的数字，顺序是 *ebdca*。

递归返回的结果见结点下面画线的数字，顺序是 *acdbe*。虚线指向的结点表示不再访问，因为前面已经被访问过。

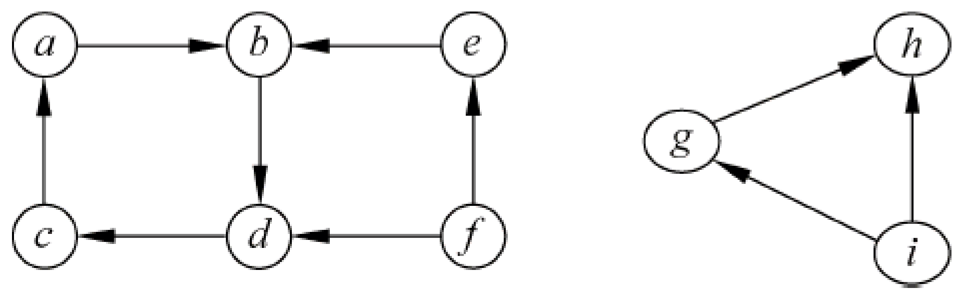


图 10.3　一个有向图例子

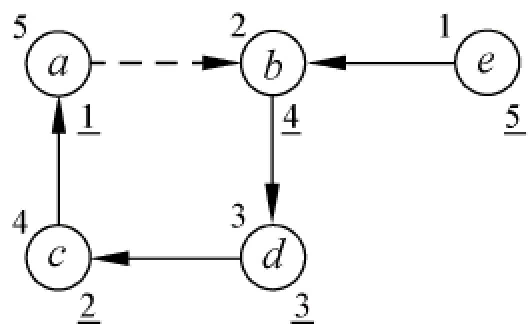


图 10.4　dfs()的访问顺序

2. 重要概念

深搜优先生成树：上面 DFS 的结果生成了一棵树，称为深搜优先生成树（depth-first spanning tree）。

树边：树上的边称为树边（tree edge）。

回退边：虚线表示的边（a,b）称为回退边（back edge），它不在树上。

在这棵树上，从起点到其他任何一个点**只有**一条路径。如果是无向图生成的树，那么任意两个点之间只有一条路径。

3. 用 dfs()处理所有点

题目经常需要处理所有的点，也可以用 dfs()实现。其思路是想象有一个虚拟结点 v，它连接了所有的点，那么在主程序中这样进行 dfs()：

```
for(int i = 0; i < n; i++)
    dfs(i);
```

读者先自己思考，写出对图 10.3 做 dfs()的过程，然后与下面的答案对照。

按字母顺序执行 dfs()，访问过程见图 10.5 结点上面的数字，顺序是 *abdcefghi*。虚线指向的结点表示不再访问。

视频讲解

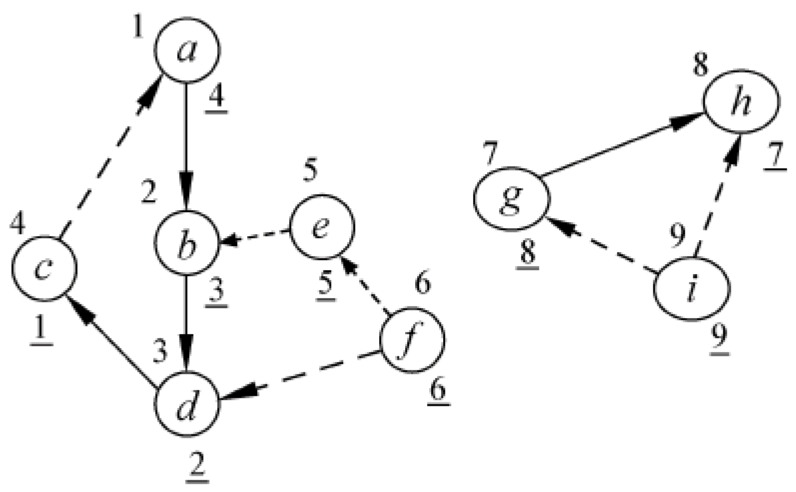


图 10.5　dfs()访问所有点的顺序

递归返回的结果见结点下面画线的数字，顺序是 *cdbaefhgi*。

请读者彻底掌握本节的内容，这是本章后面内容的基础。

10.4　拓扑排序

BFS 和 DFS 的一个直接应用是拓扑排序。

在现实生活中，人们经常要做一连串事情，这些事情之间有顺序关系或者有依赖关系，在做一件事情之前必须先做另一件事，比如安排客人的座位、穿衣服的先后、课程学习的先后等。这些事情都可以抽象为图论中的拓扑排序。

1. 拓扑排序的概念

设有 a、b、c、d 等事情，其中 a 有最高优先级，b、c 优先级相同，d 是最低优先级，表示为 $a \to (b, c) \to d$，那么 *abcd* 或 *acbd* 都是可行的排序。把事情看成图的点，把先后关系看成有向边，问题转化为在图中求一个有先后关系的排序，这就是拓扑排序，如图 10.6 所示。

显然，一个图能进行拓扑排序的充要条件是它是一个有向无环图(DAG)。有环图不能进行拓扑排序。

2. 图的入度和出度

拓扑排序需要用到点的入度和出度的概念。

出度：以点 u 为起点的边的数量称为 u 的出度。

入度：以点 v 为终点的边的数量称为 v 的入度。

一个点的入度和出度体现了这个点的先后关系。如果一个点的入度等于 0，则说明它是起点，是排在最前面的；如果它的出度等于 0，则说明它是排在最后面的。例如在图 10.7 中，点 a、c 的入度为 0，它们都是优先级最高的事情；d 的出度为 0，它的优先级最低。

拓扑排序可以有多个，例如图 10.7 中的 a 和 c，谁排在前面都可以，b 和 c 也是。

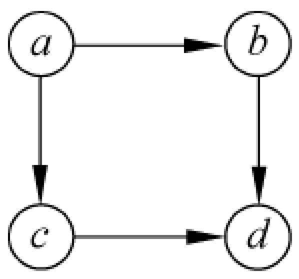


图 10.6　用图表示先后关系

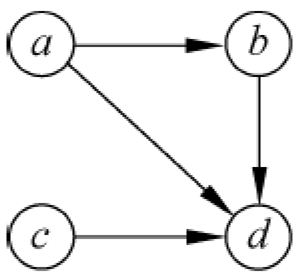


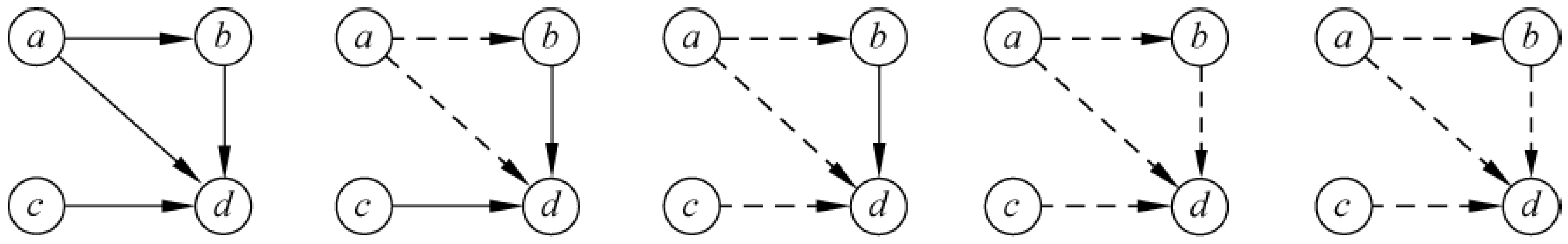
图 10.7　入度和出度

拓扑排序用 BFS 或者 DFS 都能实现。

3. 基于 BFS 的拓扑排序

基于 BFS 的拓扑排序有两种思路，即无前驱的顶点优先、无后继的顶点优先。

下面先讲解无前驱的顶点优先拓扑排序。其方法是先输出出度为 0（无前驱，优先级最高）的点，具体操作如图 10.8 所示，其中 Q 是 BFS 的队列：



(a) 进 a、c $Q=\{a,c\}$　(b) 弹出 a，进 b $Q=\{c,b\}$　(c) 弹出 c $Q=\{b\}$　(d) 弹出 b，进 d $Q=\{d\}$　(e) 弹出 d $Q=\{\}$

图 10.8　无前驱的顶点优先拓扑排序

步骤简述如下：

（1）找到所有入度为 0 的点，放进队列，作为起点，这些点谁先谁后没有关系。如果找不到入度为 0 的点，说明这个图不是 DAG，不存在拓扑排序。图 10.8(a) 中 a、c 的入度为 0，进队列。

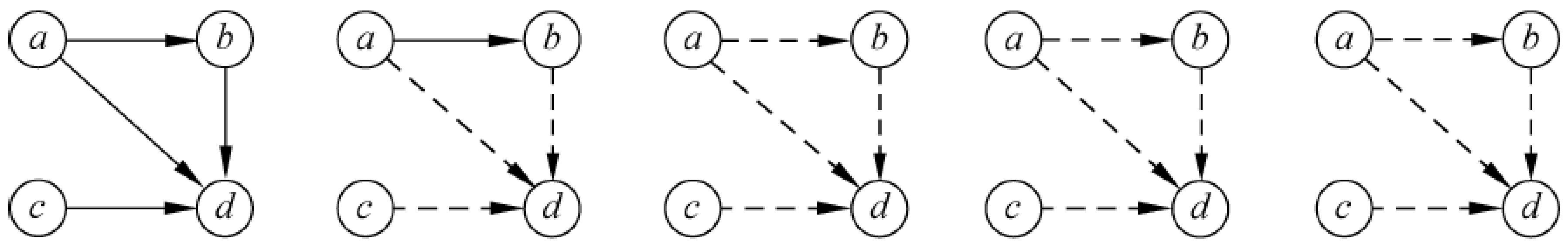
（2）弹出队首 a，a 的所有邻居点，入度减 1，把入度减为 0 的邻居点 b 放进队列，没有减为 0 的点不能放进队列。内容见图 10.8(b)。

（3）继续上述操作，直到队列为空。内容见图 10.8(c)、(d)、(e)。

队列输出 $acbd$，而且包含了所有的点，这就是一个拓扑排序。

拓扑排序无解的判断： 如果队列已空，但是还有点未进入队列，那么这些点的入度都不是 0，说明图不是 DAG，不存在拓扑排序。

以上是“无前驱”的思路。读者很容易发现，这个过程可以反过来执行，即“无后继的顶点优先”：从出度为 0（无后继，优先级最低）的点开始，逐步倒推。其示意图如图 10.9 所示，请读者自己分析过程。最后输出的是逆序 $dbca$。



(a) 进 d $Q=\{d\}$　(b) 弹出 d，进 b、c $Q=\{b,c\}$　(c) 弹出 b，进 a $Q=\{c,a\}$　(d) 弹出 c $Q=\{a\}$　(e) 弹出 a $Q=\{\}$

图 10.9　无后继的顶点优先拓扑排序

复杂度分析。在初始化时，查找入度为 0 的点，需要检查每个边，复杂度为 $O(E)$；在队列操作中，每个点进出队列一次，需要检查它直接连接的所有邻居，复杂度是 $O(V+E)$。其总复杂度是 $O(V+E)$。

4. 基于 DFS 搜索的拓扑排序

DFS 天然适合拓扑排序。

回顾 DFS 深度搜索的原理，是沿着一条路径一直搜索到最底层，然后逐层回退。这个过程正好体现了点和点的先后关系，天然符合拓扑排序的原理。事实上，在 DFS 上加一点点处理就能解决拓扑排序问题。

一个有向无环 DAG 图，如果只有一个点 u 是 0 入度的，那么从 u 开始 DFS，DFS 递归**返回**的顺序就是拓扑排序（是一个**逆序**）。DFS 递归返回的首先是最底层的点，它一定是 0 出度点，没有后续点，是拓扑排序的最后一个点；然后逐步回退，最后输出的是起点 u；输出的顺序是一个**逆序**。

以图 10.10 为例，从 a 开始，递归返回的顺序见点旁边画线的数字，即 $cdba$，是拓扑排序的逆序。

为了按正确的顺序打印出拓扑排序，编程时的处理是定义一个拓扑排序队列 list，每次递归输出的时候把它插到当前 list 的最前面，最后从头到尾打印 list，就是拓扑排序。这实际上是一个**栈，直接用** STL 的 stack < int >定义栈也行。

读者可以自己画个 DAG 图，体会 DFS 和拓扑排序的关系。

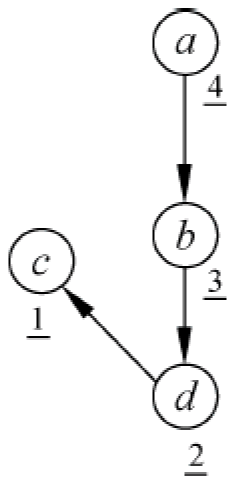


图 10.10 递归和拓扑排序

但是还有一些细节需要处理。

（1）应该以入度为 0 的点为起点开始 DFS。如何找到它？需要找到它吗？如果有多个入度为 0 的点呢？

这几个问题其实并不用特别处理。10.3 节已介绍了这个做法：**想象**有一个虚拟的点 v，它单向连接到所有其他点。这个点就是图中唯一的 0 入度点，图中所有其他的点都是它的下一层递归，而且它不会把原图变成环路。从这个虚拟点开始 DFS 就完成了拓扑排序。例如图 10.11(a)有两个 0 入度点 a 和 f；图 10.11(b)想象有个虚拟点 v，那么递归返回的顺序见点旁边画线的数字，返回的是拓扑排序的逆序。

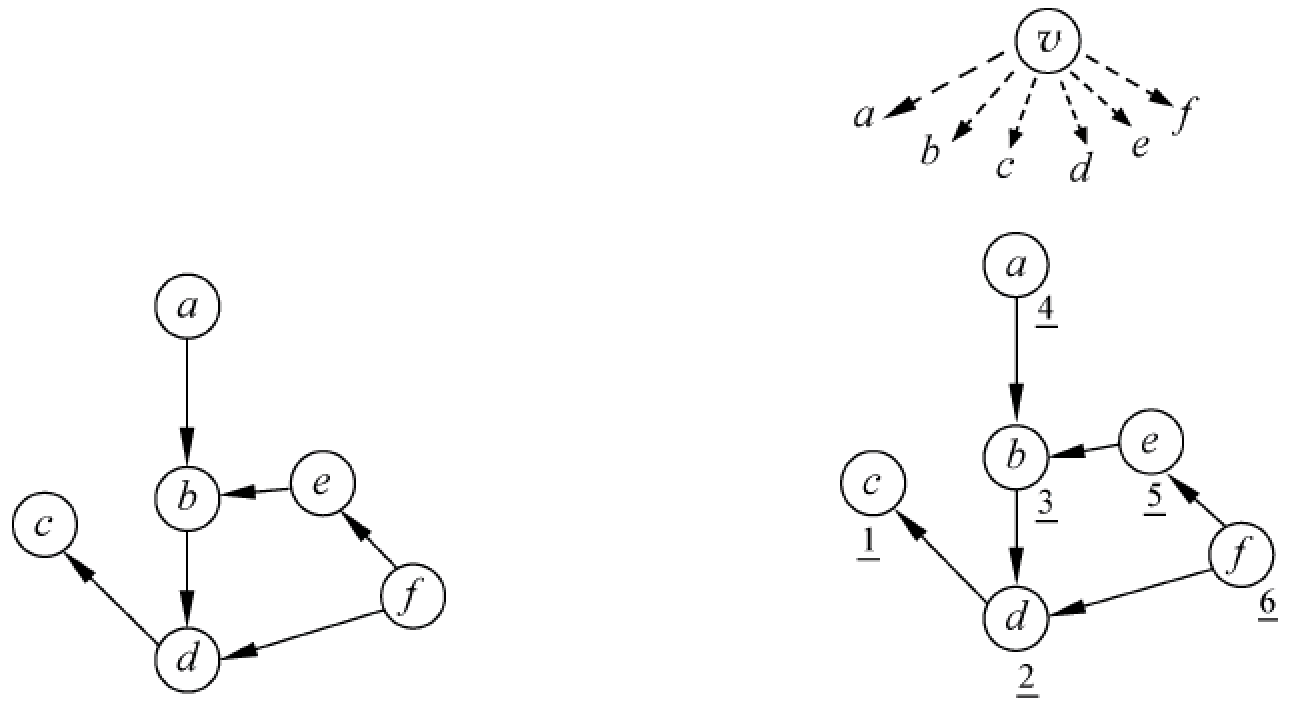


图 10.11 有多个 0 入度点的图及递归返回的顺序

在实际编程的时候并不需要处理这个虚拟点，只要在主程序中把每个点轮流执行一遍 DFS 即可。这样做相当于显式地递归了虚拟点的所有下一层点。

(2) 如果图不是 DAG，能判断吗？

图不是 DAG，说明图是有环图，不存在拓扑排序。那么在递归的时候会出现**回退边**。如果读者不理解这一点，请回顾上一节的内容。

在程序中这样发现回退边：记录每个点的状态，如果 dfs()递归到某个点时发现它仍在前面的递归中没有处理完毕，说明存在回退边，不存在拓扑排序。

5. 输出字典序最小的拓扑排序

由于一个图的拓扑排序有很多，题目一般不会要求输出所有的，而是输出字典序最小的那一个。

> **hdu 1285 “确定比赛名次”**
>
> 有 N 个比赛队进行比赛，编号依次为 $1,2,\cdots,N$，$1\leqslant N\leqslant 500$。比赛结束后，只知道每场比赛的结果。请编程确定排名。由于可能有多种结果，输出按队伍编号排序最小的那个排名。

思路很简单：在当前步骤，在所有入度为 0 的点中输出编号最小的。

先考虑用 BFS 实现。

修改 BFS 的拓扑排序程序，把普通队列改为**优先队列** Q。在 Q 中放进入度为 0 的点，每次输出编号最小的结点，然后把它的后续结点的入度减一，入度减为 0 的再放进 Q。这样就能输出一个字典序最小的拓扑排序。图 10.12 是示例。

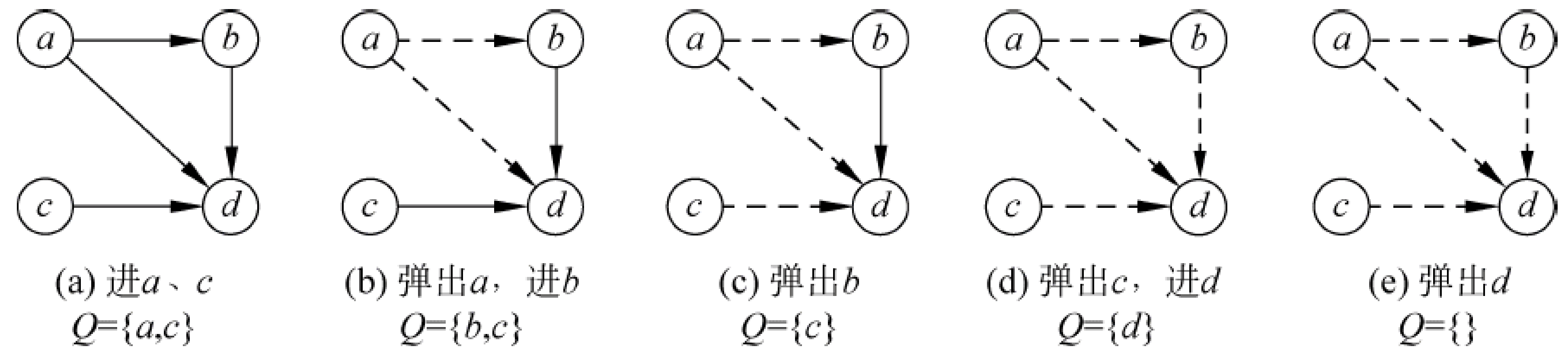


图 10.12 输出字典序的拓扑排序

如果不用优先队列找最小的点，而是用暴力查找或者排序算法，效率会比较低，读者可以试一试。

用 DFS 可以输出字典序吗？思考上述解题的过程可以发现，用 DFS 是不行的。上面处理的过程相当于把点按优先级分成不同的层次，在每个层次都要把这一层入度减为 0 的点按大小顺序输出；而 DFS 是深度搜索，处理的是上下层之间的关系，不能处理这种同层次的关系。读者可以自己画一个比较复杂的多层次的图加深理解。

【习题】

poj 1270 “Following Orders”，按字典序从小到大输出所有拓扑排序。这一题很重要。

hdu 3342 “Legal or Not”，hdu 2647 “Reward”、hdu 5695 “Gym Class”，简单拓扑排序。

hdu 4857 “逃生”,反向建图。

hdu 1811 “Rank of Tetris”,并查集+拓扑排序。

10.5 欧 拉 路

欧拉路是简单的图问题,和拓扑排序一样,也用 DFS 直接实现。

读者小时候可能玩过“一笔画”游戏:给一个图,要求一笔连续地画出整个图,必须经过每条边,并且只能经过一次,点可以重复经过。

这个问题来自于中世纪数学家欧拉的七桥问题。这条一笔画路线称为欧拉路。如果还要求起点和终点相同,则称为欧拉回路。

欧拉路:从图中某个点出发遍历整个图,图中的每条边通过且只通过一次。

欧拉回路:起点和终点相同的欧拉路。

欧拉路问题主要有两个,即是否存在欧拉路和打印出欧拉路。问题的解决主要通过处理度(degree)。一个点上连接的边的数量称为这个点的度数。在无向图中,如果度数是奇数,这个点称为奇点,否则称为偶点。在有向图中有出度和入度。

1. 欧拉路和欧拉回路是否存在

首先,图应该是连通图。在编程时用 DFS 或者并查集来判断连通性。

其次,判断图是否存在欧拉路或欧拉回路:

(1) 无向连通图的判断条件。如果图中的点全都是偶点,则存在欧拉回路;任意一点都可以作为起点和终点。如果只有两个奇点,则存在欧拉路,其中一个奇点是起点,另一个是终点。不可能出现有奇数个奇点的无向图,请读者思考。

(2) 有向连通图的判断条件。把一个点上的出度记为 1、入度记为 −1,这个点上所有的出度和入度相加就是它的度数。一个有向图存在欧拉回路,当且仅当该图所有点的度数为 0。如果只有一个度数为 1 的点、一个度数为 −1 的点,其他所有点的度数为 0,那么存在欧拉路径,其中度数为 1 的是起点、度数为 −1 的是终点。

下面用一个简单题讲解欧拉路的判断。

uva 10054 “The Necklace”

有 n 个珠子。每个珠子有两种颜色,分布在珠子的两边。一共有 50 种不同的颜色。把这些珠子串起来,要求两个相邻的珠子接触的那部分颜色相同。问是否能连成一个珠串项链?如果能,打印一种连法。

这一题是典型的无向图求欧拉回路。

首先,判断所有的点是否为偶点,如果存在奇点,则没有欧拉回路;其次,判断所给的图是否连通,不连通也不是欧拉回路。

下面的程序只判断了有无欧拉回路。关于连通性,读者可以自己用 DFS 或者并查集实现(此题很简单,没用到 DFS 和并查集)。

这一题需要注意的是可能有重边,即邻居点 u、v 之间可能有多个边。

uva 10054 题：判断欧拉回路

```
for(i = 1; i <= n; i++) {              //输入图,用邻接矩阵 G[][]存图
    scanf("%d%d", &u, &v);
    degree[u]++;
    degree[v]++;                       //记录点的度
    G[u][v]++;
    G[v][u]++;                         //0: 不连接; 1: 连接; >1: 有重边
}
for(i = 1; i <= n; i++)
    if(d[i] % 2)   break;              //存在奇点,退出; 无欧拉回路
```

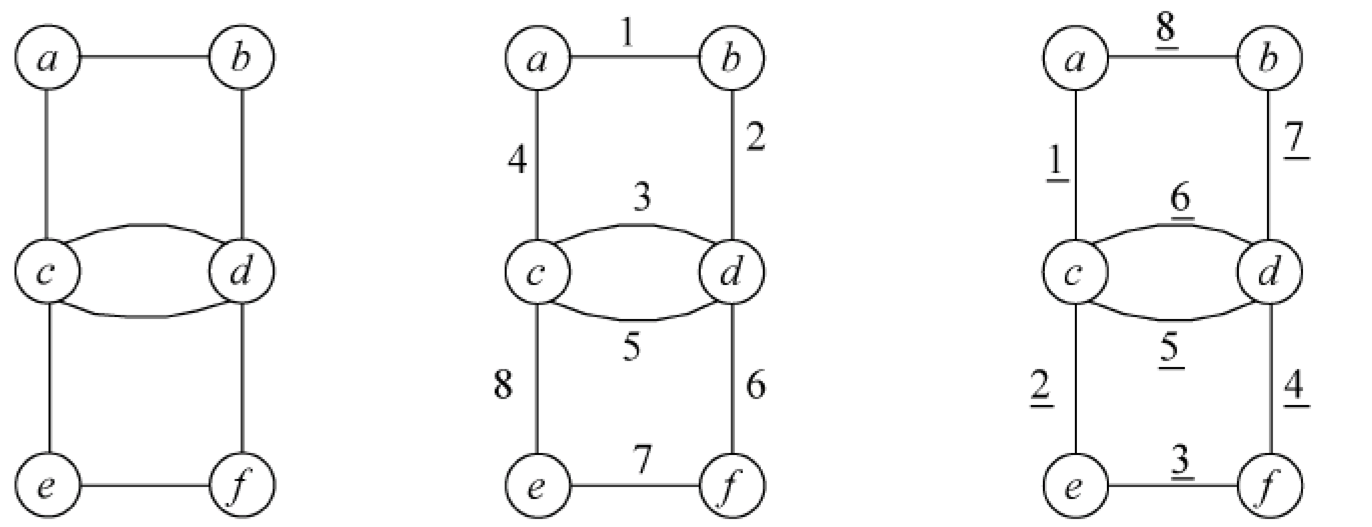
2. 输出一个欧拉回路

对一个无向连通图做 DFS 就输出了一个欧拉回路。

uva 10054 题：输出一个欧拉回路

```
void euler(int u){                     //从 u 开始 DFS
    int v;
    for(v = 1; v <= 50; v++)           //深搜 u 的所有邻居
      if(G[u][v]) {
        G[u][v]--; G[v][u]--;          //可能有重边
        euler(v);
        printf("%d %d\n", v, u);       //请思考为什么在 euler(v)后面打印
      }
}
```

图 10.13 可以帮助读者理解上面的程序。



(a) 原图　　(b) DFS访问的顺序　　(c) DFS返回的顺序

视频讲解

图 10.13　输出一个欧拉回路

从图 10.13(a)中的 a 点开始 DFS,DFS 的对象是边。图 10.13(b)边上的数字是 DFS 访问的顺序,也可以有别的顺序,图中为了帮助读者理解,特意选了一个不太“好”的顺序。图 10.13(c)边上画线的数字是 DFS 返回的顺序,它正好是一个欧拉回路。

程序中输出的路径实际上是从终点到起点的一条路径,对于无向图来说,因为起点和终点都是一个点,所以并没有关系。

如果是有向图,那么输出的是一个逆序的路径,可以用栈把逆序按正序打印出来,参考“拓扑排序”中打印路径时对栈的使用。

在上面的程序中，图是用邻接矩阵表示的。作为练习，读者可以用邻接表重写程序。

3. 用非递归 DFS 输出欧拉回路

上面用递归实现的 DFS 输出欧拉回路。递归常见的问题是爆栈，如果数据很大，就不能直接用递归，需要自己写个栈模拟递归。请读者练习下面的题目。

视频讲解

(1) poj 1780“code”。输入数字位数 n，输出一串数字，其中包含所有可能的 n 位数字序列，而且只包含一次；用字典序输出。

分析：欧拉回路问题。但是 $n=10^6$，会爆栈。

(2) hdu 4850“Wow! Such String!”。用 26 个小写字母构造一个长度为 n 的串，其中任意长度为 4 的子串都不相同；$n \leqslant 500\,000$。

分析：本题可能有 4^{26} 个子串。这一题有不同解法，如果用 DFS，也容易爆栈。

4. 混合图欧拉路问题

有的图不是单纯的有向图或无向图，而是二者的混合，同时存在有向边和无向边。这是一个比较困难的问题，需要用最大流求解，具体内容见“10.11.3 Dinic 算法和 ISAP 算法”。

【习题】

hdu 1878 “欧拉回路”。判断是否存在回路，无向图。

hdu 1116 “Play on Words”。首尾连单词，有向图，可以分别用 DFS 和并查集判断连通性。

hdu 5883 “The Best Path”。无向图欧拉路。

10.6 无向图的连通性

10.6.1 割点和割边

在无向图中，所有能互通的点组成了一个“连通分量”。在一个连通分量中有一些关键的点，如果删除它，会把这个连通分量分成两个或更多，这种点称为割点(Cut vertex)。

类似的有割边(Cut edge，又称为桥，bridge)问题。在一个连通分量中，如果删除一个边，把这个连通分量分成了两个，这个边称为割边。

研究割点和割边是很有意义的。从割点、割边扩展出**双连通**问题，即如何实现一个没有割点和割边的图。例如在计算机网络中，可靠性是重要的问题，希望能在某些网络结点出故障的情况下不影响整个网络的通畅。那么，应该如何布置网络才能不出现割点，并且部署的结点最少？

本节先研究一个基本问题：在一个无向连通图 G 中有多少个割点？

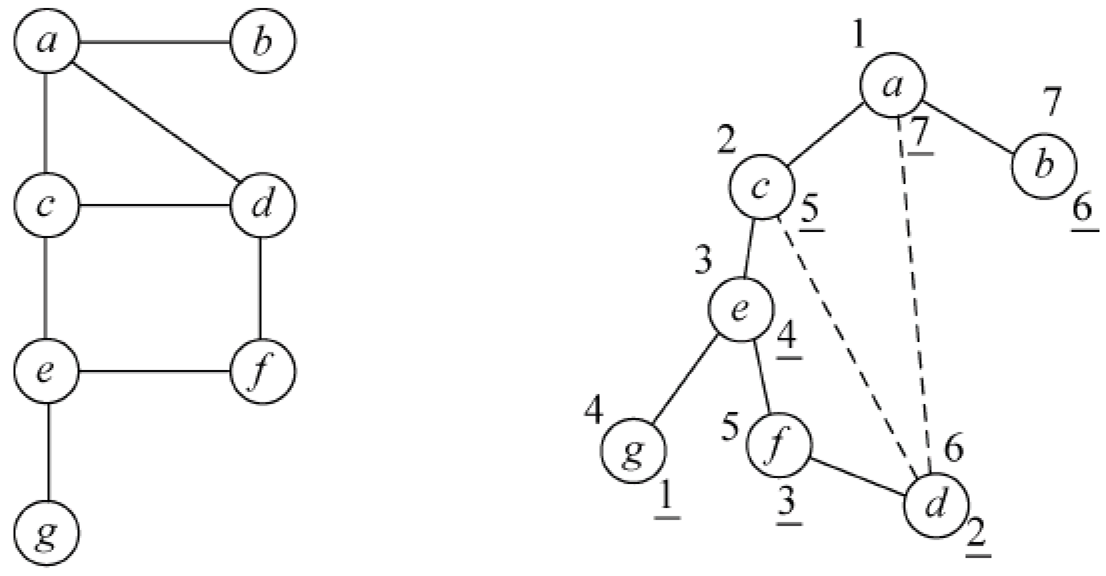
暴力方法：删除每个点，然后用 DFS 求连通性，如果连通分量变多，那么就是割点。其复杂度是 $O(V(V+E))$，不是好算法。

下面介绍用 DFS 求割点的算法，即利用“深搜优先生成树”求割点。请读者先回顾 10.3 节的概念。

在一个连通分量 G 中，对任意一个点 s 做 DFS，能访问到所有点，产生一棵"深搜优先生成树"T。那么对 G 求割点，和 T 有什么关系呢？

定理 10.1：T 的根结点 s 是割点，当且仅当 s 有两个或更多的子结点。这个定理很容易理解，如果 s 是割点，它会把图分成不相连的几部分，这几个部分都会生成子树；如果 s 不是割点，它只会连接一个子树。

读者可以验证图 10.14。图(b)是 a 的生成树，a 点是割点，它有子结点 b 和 c。图中点上面的数字是递归的顺序，下面画线的数字是递归返回的顺序。



(a) 原图　　(b) 点 a 的生成树

图 10.14　根结点是割点的判断

b 不是割点，如果用 b 生成树，只有一个子结点 a。

定理 10.2：T 的非根结点 u 是割点，当且仅当 u 存在一个子结点 v，v 及其后代都没有回退边连回 u 的祖先。这个定理也容易理解，如果 u 是割点，它会把图分成两部分或更多，其中至少一个后代肯定没有通过其他边(回退边，即绕过 u 回去的边)连回 u 的祖先，否则图就不会被分开了。

例如图 10.14(b)中的 c 点，它的子结点只有一个 e，而 e 后面有个子结点 d 有回退边连回了根结点 a，所以 c 不是割点。再看 e 点，有一个子结点 g，没有回退边连回 e 的祖先，所以 e 是割点。

如何编程实现定理 10.2？

设 u 的一个直接后代是 v。

定义 num[u]，记录 DFS 对每个点的访问顺序，num 值随着递推深度增加而变大。

定义 low[v]，记录 v 和 v 的后代能连回到的祖先的 num。

只要 low[v]≥num[u]，就说明在 v 这个支路上没有回退边连回 u 的祖先，最多退到 u 本身。这就是定理 10.2。

下面的图 10.15 是例子，low[u]的初始值等于 num[u]，即连到自己。图 10.15(a)没有回退边。b 的后代是 c，low[c]=3，num[b]=2，有 low[c]≥num[b]，说明 b 的支路 c 上没有回退边连回去，所以 b 是割点。

在图 10.15(b)中，观察 low[]是如何更新的。最后访问的 d 是递归最深处的点，它的 num[d]=4，它有回退边连到 b，low[d]的初始值是 4，更新为 low[d]=num[b]=2，表示有回退边到 b。然后 d 递归回到 c，low[c]更新为 low[c]=low[d]=2，表示 c 通过后代能回退到 b。以上是 low[]的更新过程。继续考察 c：由于 low[d]=2，num[c]=3，说明 c 的后代 d 有回退边连到了 c 的祖先，所以 c 不是割点。

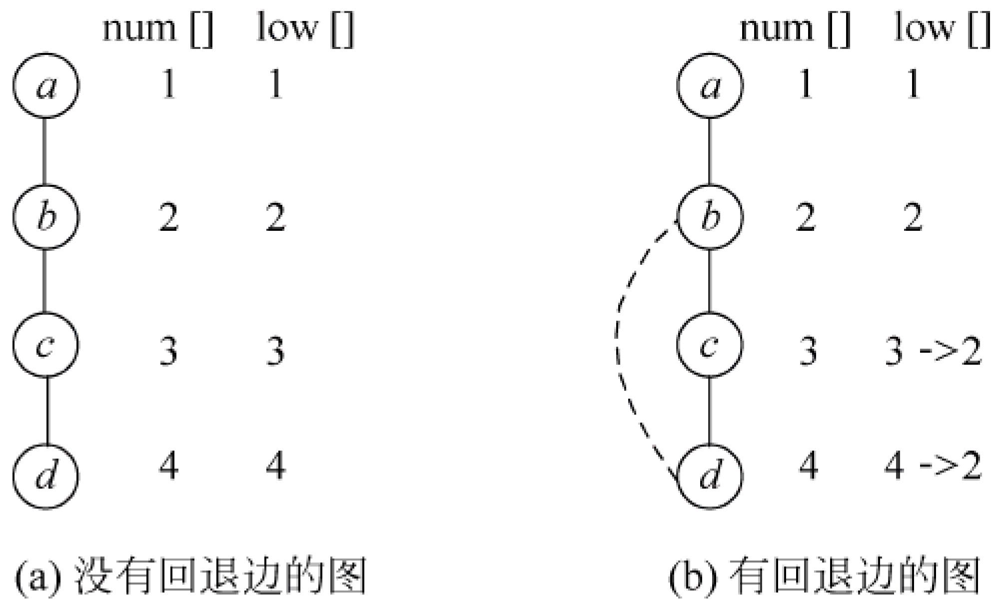


图 10.15　非根结点是割点的判断

特别有意思的是，上述判断割点的条件 low[v]≥num[u]只要改为 low[v]>num[u]就能用于**判断割边**。这表示 u 的支路 v 以及 v 的后代只能回退到 v，而到不了 u，那么边(u,v)肯定就是割边。例如图 10.15(b)中的 b 点，有 low[c]=2，num[b]=2，说明(b,c)不是割边；再看 a 点，有 low[b]=2，num[a]=1，low[b]>num[a]，所以(a,b)是割边。

poj 1144 "network"

输入一个无向图，求割点的数量。

一个电话线公司正在建立一个电话线缆网络。他们用线缆连接了若干个地点，这些线是双向的，每个地点都有一个电话交换机。从每个地点都能通过线缆到达其他任意的地点，并不一定直接连接，可以通过若干个交换机来到达目的地。有时候某个地点供电出问题，交换机会停止工作。工作人员意识到，除非这个地点是不可达的，否则它还会导致一些其他的地点不能互相通信。称这个地点为关键点。工作人员想要写一个程序找到所有关键点的数量。

在下面的程序中，用 int dfn 记录进入递归的顺序(也称为时间戳)，然后赋值给这个递归中点 u 的 num[u]。

poj 1144 部分代码

```
const int N = 109;
int low[N], num[N], dfn;                  //dfn 记录递归的顺序,用于给 num 赋值
bool iscut[N];
vector < int > G[N];                      //存图
void dfs(int u, int fa){                  //u 的父结点是 fa
low[u] = num[u] = ++ dfn;                 //初始值
int child = 0;                            //孩子数目
for (int i = 0;i < G[u].size(); i++){     //处理 u 的所有子结点
    int v = G[u][i];
    if (!num[v]) {                        //v 没访问过
        child++;
        dfs(v, u);
        low[u] = min(low[v], low[u]);     //用后代的返回值更新 low 值
        if (low[v] > = num[u] && u != 1)
            iscut[u] = true;              //标记割点
```

```
        }
        else if(num[v] < num[u] && v != fa)
                    //处理回退边,注意这里 v != fa,fa 是 u 的父结点
                    //fa 也是 u 的邻居,但是前面已经访问过,不需要处理它
            low[u] = min(low[u], num[v]);
    }
    if (u == 1 && child >= 2)                   //根结点,有两个以上不相连的子树
        iscut[1] = true;
    }
    int main(){
        int ans, n;
        //在这里输入图,程序略
        memset(low, 0, sizeof(low));
        memset(num, 0, sizeof(num));
        dfn = 0;
        memset(iscut, false, sizeof(iscut));
        ans = 0;
        dfs(1, -1);                             //DFS 的起点是 1
        for(int i = 1;i <= n;i++)  ans += iscut[i];
        printf("%d\n", ans);
    }
```

判断割边。把程序中的 if (low[v] >= num[u] && u !=1)改为 if (low[v]>num[u] && u !=1),其他程序不变,就是求割边的数量。

10.6.2 双连通分量

在一个连通图中选任意两点,如果它们之间至少存在两条"点不重复"的路径,称为点双连通。一个图中的点双连通极大子图称为"点双连通分量"(block,或者 2-connected component)。点双连通分量是一个"可靠"的图,去掉任意一个点,其他点仍然是连通的。也就是说,点双连通分量中没有割点。

类似地有"边双连通分量",如果任意两点之间至少存在两条"边不重复"的路径,称为"边双连通"。在边双连通图中去掉任意一个边,图仍然是连通的。也就是说,边双连通图中没有割边。

1. 点双连通分量

在一个无向图 G 中有多少个点双连通分量?

求解点双连通分量和求割点密切相关。不同的点双连通分量最多只有一个公共点,即某一个割点;任意一个割点都是至少两个点双连通分量的公共点。

计算点双连通分量一般用 Tarjan 算法[①],下面是算法的思路。

前面讲解了如何用 DFS 进行割点的计算,可以发现,在找到一个割点的时候已经完成了一次对某个极大点双连通子图的访问。那么,在进行 DFS 的过程中,把遍历过的点保存起来,就可以得到这个点双连通分量。

① Tarjan 提出了很多算法,这是其中之一。

DFS 的访问过程用栈来保存是最合理的，所以，在求解割点的过程中，用一个栈保存遍历过的边，然后每当找到一个割点，即满足关系 low[v] >= num[u]的点 u，就将栈里的边拿出来。

注意，放入栈中的不是点，而是边。因为一个边只属于一个点双连通分量，而一个割点属于多个点双连通分量，如果进入栈中的是点，这个割点弹出来之后就只能给一个点双连通分量了，它连接的其他点双连通分量就会少了这个点。

练习题：poj 1523 "SPF"，一个图中有多少个割点？每个割点能把网络分成几个点双连通分量？

2. 边双连通分量

给定一个图 G，它有多少个边双连通分量？至少应该添加多少条边，才能使任意两个边双连通分量之间都是双连通的，也就是使图 G 是双连通的？

poj 3352 "Road Construction"

给定一个无向图 G，图中没有重边。问添加几条边才能使无向图变成边双连通图。

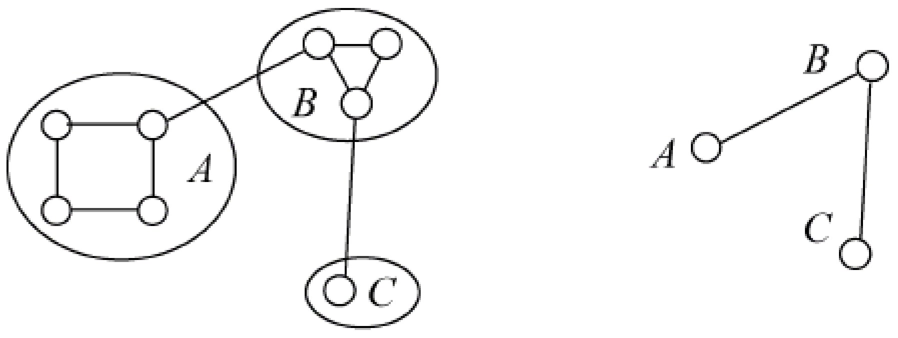
边双连通分量的计算用到了"缩点"的技术。

(1) 首先找出图 G 的所有边双连通分量。

在 DFS 过程中，图 G 所有的点都生成一个 low 值，low 值相同的点必定在同一个边双连通分量中。DFS 结束后，有多少 low 值就有多少个边双连通分量。

(2) 把每一个边双连通分量都看作一个点，即把那些 low 值相同的点合并为一个"缩点"。这些缩点形成了一棵树，例如图 10.16。

(3) 问题被转化为：至少在缩点树上增加多少条边才能使这棵树变为一个边双连通图。容易推导出：至少增加的边数＝(总度数为 1 的结点数＋1)/2。例如图 10.16(b)有两个度数为 1 的点 A、C，至少增加的边数＝(2＋1)/2＝1。



(a) 连通分量 (b) 缩点图

图 10.16 边双连通分量的缩点

poj 3352 程序

```
#include <cstring>
#include <vector>
#include <stdio.h>
using namespace std;
const int N = 1005;
int n, m, low[N], dfn;
vector<int> G[N];                         //存图
void dfs(int u,int fa){                   //计算每个点的 low 值
   low[u] = ++dfn;
   for(int i=0;i<G[u].size();i++){
       int v = G[u][i];
       if(v == fa) continue;
       if(!low[v])
```

```
            dfs(v,u);
        low[u] = min(low[u], low[v]);
    }
}
int tarjan(){
    int degree[N];                          //计算每个缩点的度数
    memset(degree,0,sizeof(degree));
    for(int i=1; i<=n; i++)                 //把有相同 low 值的点看成一个缩点
        for(int j=0; j<G[i].size(); j++)
            if(low[i] != low[G[i][j]])
                degree[low[i]]++;
    int res=0;
    for(int i=1;i<=n;i++)                   //统计度数为 1 的缩点的个数
        if(degree[i]==1) res++;
    return res;
}
int main(){
    while(~scanf("%d%d", &n, &m)){
        memset(low, 0, sizeof(low));
        for(int i=0; i<=n; i++)   G[i].clear();
        for(int i=1; i<=m; i++){
            int a, b;
            scanf("%d%d", &a, &b);
            G[a].push_back(b);  G[b].push_back(a);
        }
        dfn = 0;
        dfs(1,-1);
        int ans = tarjan();
        printf("%d\n",(ans+1)/2);
    }
    return 0;
}
```

【习题】

hdu 3394 “Railway”,点双连通分量。

hdu 3749 “Financial Crisis”,点双连通分量。

hdu 2460 “Network”,边双连通分量。

hdu 4587 “TWO NODES”,无向图求割点。

10.7 有向图的连通性

本节的内容与拓扑排序的思想有关,读者在阅读之前请先认真学习本章“10.4 拓扑排序”的内容。

强连通。在有向图 G 中,如果两个点 u、v 是互相可达的,即从 u 出发可以到达 v,从 v

出发也能到达 u，则称 u 和 v 是强连通的。如果 G 中的任意两个点都是互相可达的，称 G 是强连通图。

强连通分量。如果一个有向图 G 不是强连通图，那么可以把它分成多个子图，其中每个子图的内部是强连通的，而且这些子图已经扩展到最大，不能与子图外的任意点强连通，称这样的一个"极大强连通"子图是 G 的一个强连通分量(Strongly Connected Component，SCC)。

一个常见的问题：G 中有多少个 SCC？在解决这个问题之前需要研究 SCC 的特征。

(1) 出度和入度。一个点必须有出发的边，也有到达的边，这样才会与其他点强连通。

(2) 把一个 SCC 从 G 中挖掉，不影响其他点的强连通性。可以把图上的一个个 SCC 想象成一个个岛，岛内部是强连通的；岛之间只有单向道路连接，不会形成环路。把每个岛虚拟成一个点，那么所有这些虚拟点构成的虚拟图是一个有向无环图 DAG；这个虚拟 DAG 图中的点与其他点都不是强连通的，DAG 中的虚拟点的数量就是 SCC 的数量，如图 10.17 所示。可以推论出，每个岛都可以挖掉，而不会影响其他岛内部的连通性。

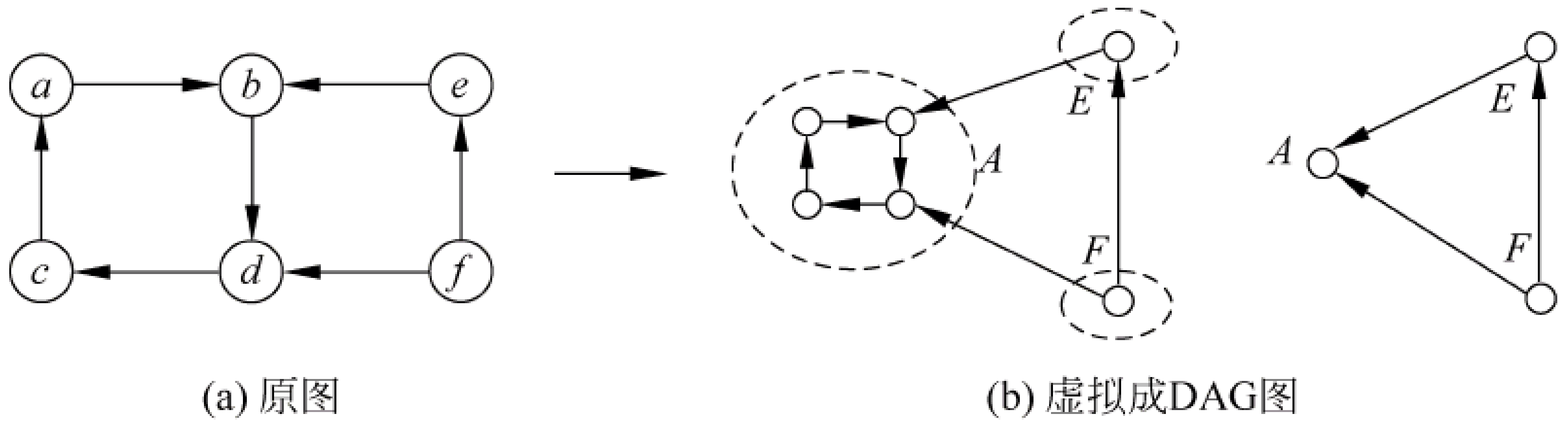


图 10.17　SCC 的虚拟图

用暴力的方法求 SCC 是对每个点求连通性，然后进行比较，那些互相连通的点就组成了 SCC。这可以通过对每个点都进行 DFS 或者 BFS 搜索得到，例如对图 10.17(a)进行搜索的结果如下：

分别从 a、b、c、d 点出发，可以到达：$\{a,b,c,d\}$；

从 e 点出发可以到达：$\{a,b,c,d,e\}$；

从 f 点出发可以到达：$\{a,b,c,d,e,f\}$。

最少的 $\{a,b,c,d\}$ 是一个强连通分量，从整个图中挖掉它，剩下最小的是 $\{e\}$，再挖掉它，最后是 $\{f\}$，得到 3 个 SCC，即 $\{a,b,c,d\}$、$\{e\}$、$\{f\}$。

暴力法的复杂度是 $O(V^2+E)$。

求 SCC 有 3 种高效算法，即 Kosaraju、Tarjan、Garbow，它们的复杂度都是 $O(V+E)$，但 Kosaraju 要差一些。下面介绍 Kosaraju、Tarjan 算法。

10.7.1　Kosaraju 算法

Kosaraju 算法用到了"反图"的技术，基于下面两个原理：

(1) 一个有向图 G，把 G 所有的边反向，建立反图 rG，反图 rG 不会改变原图 G 的强连通性。也就是说，图 G 的 SCC 数量与 rG 的 SCC 数量相同。这里直接用上面的虚拟 DAG 图做例子，图 10.18(a)中的 A、E、F 是 3 个 SCC，内部的点都是强连通的。

(2) 对原图 G 和反图 rG 各做一次 DFS，可以确定 SCC 数量。

对原图 G 做 DFS 是为了确定点的先后顺序。可以发现，对生成的虚拟 DAG 图，可以用 DFS 做拓扑排序，排序结果是 F、E、A（不过，此时并没有确定哪些点是属于 A、E、F 的）。而且，F 内部优先级最高的那个点，优先级高于 E、A 内部所有的点；E 内部优先级最高的那个点，优先级高于 A 内部所有的点。这个有用的结果将用于下面的步骤。

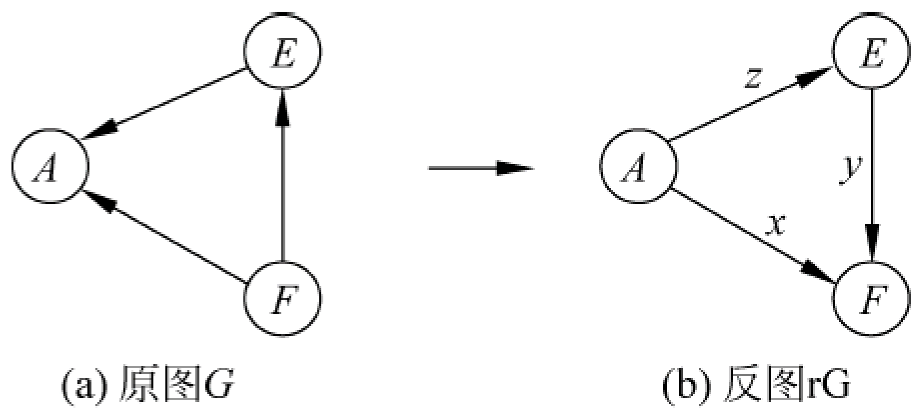


图 10.18　原图与反图

确定了顺序，然后从优先级最高的点（这个点属于 F）开始，在反图上做 DFS。为什么要在反图上做 DFS？这样做可以求得被隔离的“岛”。例如求 F 包含哪些点，想办法把 F 和其他点隔离就好了；原图中 F 是只有出度的点，改成反图后，F 变成了只有入度的点，那么从 F 出发做 DFS，就会被反边 **x、y 堵住**，DFS 搜索到的点被限制在 F 内。显然，只能搜到并且能全部搜到 F 内部的点，而无法到达 A、E，这样就确定了 F，也就是确定了第 1 个 SCC。

下一步，删除 F，然后继续在剩下的优先级最高的点开始搜，这一步搜到的点属于 E，而 E 也被反边 **z 堵住**，只能搜到属于 E 的点，确定了第 2 个 SCC。最后，删除 E，确定属于 A 的点，也就是确定了第 3 个 SCC。

算法步骤如下：

（1）在 G 上做一次 DFS，标记点的先后顺序。在 DFS 的过程中标记所有经过的点，把递归到最底层的那个点标记为最小，然后在回退的过程中，其他点的标记逐个递增。和上节拓扑排序中的 DFS 操作一样，并不需要找一个特殊的点作为起点，可以想象有一个起点 v，v 连接所有的结点，从 v 开始 DFS。

在图 10.19(a)中，从虚拟的点 v 出发，按 a、b、c、d、e、f 的顺序执行 DFS，DFS 返回的结果是 c、d、b、a、e、f；每个点的大小标记见图中的数字。如果搜索顺序不同，结果也会不同；但是，不管是什么顺序，**f 的标记肯定最大**，这是拓扑排序的原理。读者可以试试其他顺序，验证这个结论。

视频讲解

（2）在反图 rG 上再做一次 DFS，顺序从标记最大的点开始到最小的点。首先是点 f，记录所有它能到达的点，这些点组成了第 1 个 SCC，图 10.19(b)中点 f 只能到达自己，这是第 1 个 SCC；然后删除 f，从剩下的最大的点继续 DFS，这次是点 e，是第 2 个 SCC；最后从点 a 开始搜，返回$\{c,d,b,a\}$，这是第 3 个 SCC。

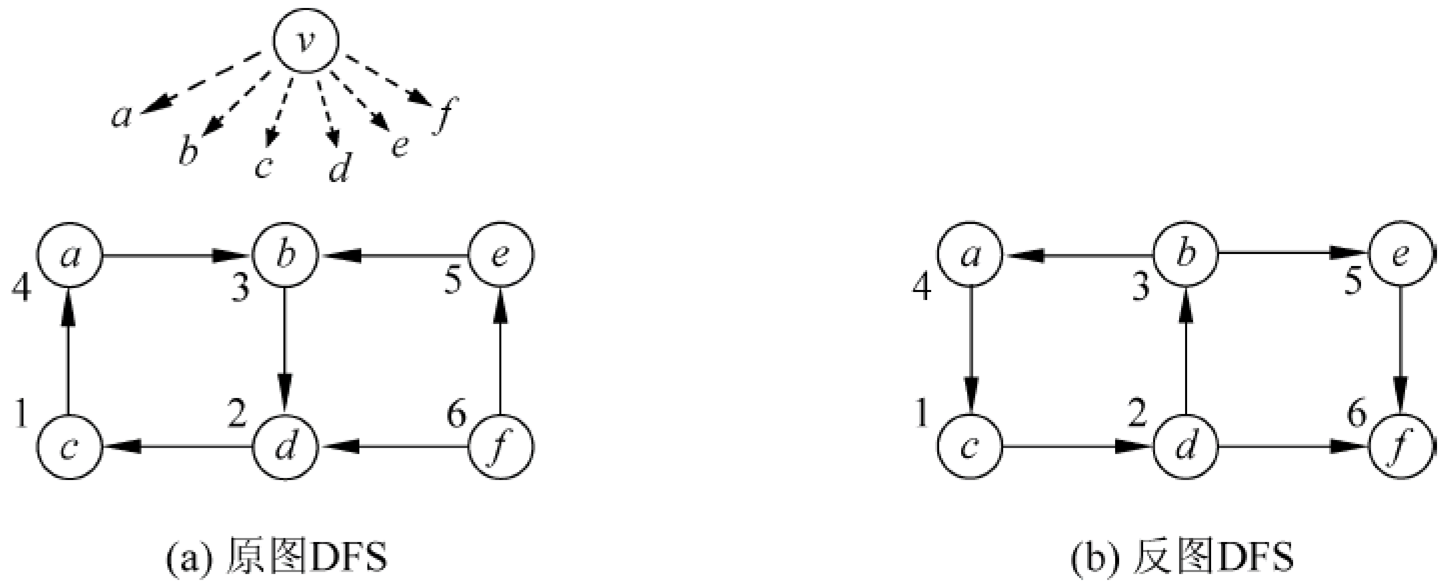


图 10.19　原图和反图的 DFS

> **hdu 1269 “迷宫城堡”**
>
> 一个有向图，有 n 个点（$n \leqslant 10\ 000$）和 m 条边（$m \leqslant 100\ 000$）。判断整个图是否强连通，如果是，输出 Yes，否则输出 No。

hdu 1269 的 Kosaraju 算法代码[①]

```
include <bits/stdc++.h>
using namespace std;
const int NUM = 10005;
vector<int> G[NUM], rG[NUM];
vector<int> S;                              //存第一次 dfs1()的结果：标记点的先后顺序
int vis[NUM], sccno[NUM], cnt;              //cnt: 强连通分量的个数
void dfs1(int u) {
    if(vis[u]) return;
    vis[u] = 1;
    for(int i = 0; i < G[u].size(); i++)  dfs1(G[u][i]);
    S.push_back(u);                         //记录点的先后顺序,标记大的放在 S 的后面
}
void dfs2(int u) {
    if(sccno[u]) return;
    sccno[u] = cnt;
    for(int i = 0; i < rG[u].size(); i++)  dfs2(rG[u][i]);
}
void Kosaraju(int n) {
    cnt = 0;
    S.clear();
    memset(sccno, 0, sizeof(sccno));
    memset(vis, 0, sizeof(vis));
    for(int i = 1; i <= n; i++)  dfs1(i);  //点的编号：1～n.递归所有点
    for(int i = n-1; i >= 0; i-- )
        if(!sccno[S[i]]) {cnt++; dfs2(S[i]);}
}
int main(){
    int n, m, u, v;
    while(scanf("%d%d", &n, &m), n != 0 || m != 0) {
        for(int i = 0; i < n; i++) {G[i].clear(); rG[i].clear();}
        for(int i = 0; i < m; i++){
            scanf("%d%d", &u, &v);
            G[u].push_back(v);              //原图
            rG[v].push_back(u);             //反图
        }
        Kosaraju(n);
        printf("%s\n", cnt == 1 ? "Yes" : "No");
    }
    return 0;
}
```

① 部分代码参考《算法竞赛入门经典训练指南》，刘汝佳，清华大学出版社，320 页。

该程序用 cnt 记录 SCC 的数量，并且统计了每个点所属的 SCC，sccno[i]是第 i 个点所属的 SCC。在 dfs2()中，sccno[i]也被用于记录点 i 是否被访问，如果 sccno[i]不等于 0，说明它已经被处理过；在 dfs1()中，用 vis[i]记录点 i 是否被访问。

Kosaraju 算法的复杂度是 $O(V+E)$。

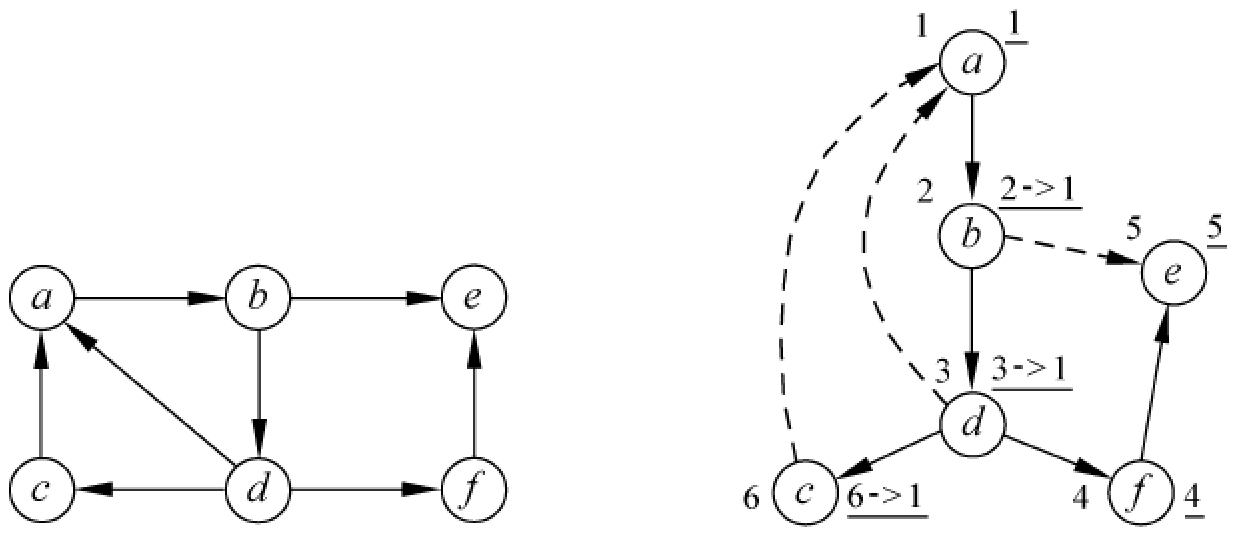
10.7.2 Tarjan 算法

上面的 Kosaraju 算法，其做法是从图中一个一个地把 SCC“挖”出来。Tarjan 算法能在一次 DFS 中把所有点都按 SCC 分开。这并不是不可思议的，它利用了 SCC 的如下特点。

定理 10.3：一个 SCC，从其中任何一个点出发，都至少有一条路径能绕回到自己。

在继续讲解之前，请读者先回顾无向图 DFS 中求割点的 low[]和 num[]操作。Tarjan 算法用到了同样的技术，这个技术结合定理 10.3 就是 Tarjan 算法。

下面是例子，图 10.20 中有 3 个 SCC，即{a,b,d,c}、{e}、{f}。



(a) 原图　　(b) 对图做DFS

图 10.20　SCC 的 low[]和 num[]操作

图 10.20(a)是原图。图(b)对它做 DFS，每个点左边的数字标记了 DFS 访问它的顺序，即 num[]值，右边的画线数字是 low[]值，即能返回到的最远祖先。每个点的 low[]初始值等于 num[]，即连到自己。观察 c 的 low[]值是如何更新的：它的初始值是 6，然后有一个回退边到 a，所以更新为 1；它的递归祖先 d、b 的 low[]值也跟着更新为 1。e 和 f 的 low[]值不能更新。

图 10.20(b)是从 a 开始 DFS 的，a 成为{a,b,d,c}这个 SCC 的共同祖先。其实，从{a，b,d,c}中任意一个点开始 DFS，这个点都会成为这个 SCC 的祖先。认识到这些，可以帮助读者理解后面的解释：可以用栈分离不同的 SCC。

图 10.20(b)中的 low[]值有 3 个部分，即等于 1 的{a,b,d,c}、等于 4 的{f}、等于 5 的{e}。这就是 3 个 SCC。

完成以上步骤，似乎已经解决了问题。每个点都有了自己的 low[]值，相同 low[]值的点属于一个 SCC。那么只要再对所有点做一个查询，按 low[]值分开就行了，其复杂度是 $O(V)$。其实有更好的办法，即在 DFS 的同时把点按 SCC(有相同的 low[]值)分开。

以图 10.21 为例，其中有 3 个 SCC，即 A、E、F。假设从 F 中的一个点开始 DFS，DFS 过程可能会中途跳出 F，转入 A

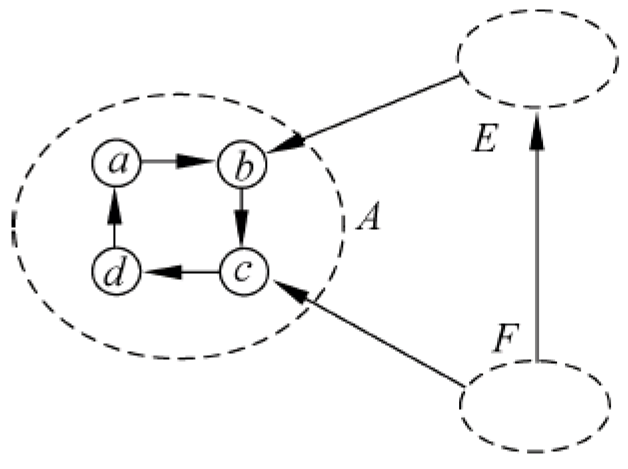


图 10.21　把图分成多个 SCC

或者 E，总之，最后会进入一个 SCC。

(1) 假设 DFS 过程是 $F \to E \to A$，最后进入 A。

(2) 在 A 这个 SCC 中将完成 A 内所有点的 DFS 过程，也就是说，最后的几步 DFS 会集中在 A 中的点 a、b、c、d。这几个点会计算得到相同的 low[]值，标记为一个 SCC，这样就好了。

(3) DFS 递归从 A 回到 E，并在 E 中完成 E 内部点的 DFS 过程。

(4) 回到 F，在 F 内完成递归过程。

以上过程如何编程？读者能想起来，DFS 搜索是用递归实现的，而递归和栈这种数据结构在本质上是一致的。所以，可以**用栈来帮助处理**：

(1) 从 F 开始递归搜索，访问到的某些点进入栈；

(2) E 中的某些点进入栈；

(3) 在 DFS 的最底层，A 的所有点将被访问到并进入栈，当前栈顶的几个元素就是 A 的点，标记为同一个 SCC，并弹出栈；

(4) DFS 回到 E，在 E 中完成所有点的搜索并且入栈，当前栈顶的几个元素就是 E 的点，标记为同一个 SCC，并弹出栈；

(5) 回到 F，完成 F 的所有点的搜索并且入栈，当前栈顶的几个元素就是 F 的点，标记为同一个 SCC，并弹出栈。结束。

为加深对上述过程中栈的理解，读者可以思考最先进入栈的点。每进入一个新的 SCC，访问并入栈的第一个点都是这个 SCC 的祖先，它的 num[]值等于 low[]值，这个 SCC 中所有点的 low[]值都等于它。

仍然以 hdu 1269 题为例，给出 Tarjan 算法代码。程序中用一个数组 int stack[N]模拟栈，读者可以尝试直接用 STL 的 stack <int>定义栈。

hdu 1269 的 Tarjan 算法代码

```
include <bits/stdc++.h>
using namespace std;
const int N = 10005;
int cnt;                                    //强连通分量的个数
int low[N], num[N], dfn;
int sccno[N], stack[N], top;                //用 stack[ ]处理栈,top 是栈顶
vector<int> G[N];
void dfs(int u){
    stack[top++] = u;                       //u 进栈
    low[u] = num[u] = ++dfn;
    for(int i = 0; i < G[u].size(); ++i){
        int v = G[u][i];
        if(!num[v]){                        //未访问过的点,继续 DFS
            dfs(v);                         //DFS 的最底层,是最后一个 SCC
            low[u] = min(low[v], low[u]);
        }
        else if(!sccno[v])                  //处理回退边
            low[u] = min(low[u], num[v]);
    }
    if(low[u] == num[u]){                   //栈底的点是 SCC 的祖先,它的 low = num
```

```
            cnt++;
            while(1){
                int v = stack[--top];          //v 弹出栈
                sccno[v] = cnt;
                if(u==v) break;                //栈底的点是 SCC 的祖先
            }
        }
    }
    void Tarjan(int n){
          cnt = top = dfn = 0;
          memset(sccno,0,sizeof(sccno));
          memset(num,0,sizeof(num));
          memset(low,0,sizeof(low));
          for(int i=1; i<=n; i++)
              if(!num[i])
                dfs(i);
    }
    int main(){
        int n,m,u,v;
        while(scanf("%d%d", &n, &m), n != 0 || m != 0) {
            for(int i=1; i<=n; i++){G[i].clear();}
            for(int i=0; i<m; i++){
                scanf("%d%d", &u, &v);
                G[u].push_back(v);
            }
            Tarjan(n);
            printf("%s\n", cnt == 1 ? "Yes" : "No");
        }
        return 0;
    }
```

Tarjan 算法的复杂度也是 $O(V+E)$，但是它只做了一次 DFS，比 Kosaraju 算法快。

【习题】

hdu 1827 "Summer Holiday"，Tarjan 缩点。

hdu 3072 "Intelligence System"，Tarjan+贪心。

hdu 3836 "Equivalent Sets"，给定有向图，至少要添加多少条边才能成为强连通图？

hdu 3639 "Hawk-and-Chicken"，强连通分量+缩点。

hdu 3861 "The King's Problem"，Tarjan+最小路径覆盖。

hdu 1530 "Maximum Clique"，最大团简单题目。强连通分量的一个应用是最大团问题(Maximum Clique Problem，MCP)。

10.8 2-SAT 问题

2-SAT 问题可以用强连通分量和拓扑排序解决。

先用一个例子说明什么是 2-SAT 问题。

hdu 3062 “Party”

有 n 对夫妻被邀请参加一个聚会，每对夫妻中只有 1 人可以列席。在 $2n$ 个人中，某些人(不包括夫妻)之间有着很大的矛盾，有矛盾的两个人不会同时出现在聚会上。问有没有可能让 n 个人同时列席？

1. 数字逻辑的解法

视频讲解

如果学过计算机系的大二课程“数字逻辑”，可以用卡诺图帮助理解这个题目。

输入样例：有 3 对夫妻 A(包括 A 男和 A 女，B 和 C 也是)、B、C，其中 A 男和 B 女有矛盾，A 女和 C 女有矛盾，A 男和 C 男有矛盾。

输出：所有合法的出席情况。

分析如下：

(1) 夫妻不同时出席。例如，第 A 对夫妻，丈夫是 A，妻子是 $\overline{A}$，因为夫妻不同时出席，所以互为反变量①。

(2) 不同夫妻的限制条件。例如，A 男和 B 女(B 女用 $\overline{B}$ 表示)有矛盾，即 A 和 $\overline{B}$ 不会同时出现，有 $A\overline{B}=0$。一共有 3 个限制：$A\overline{B}=0$，$\overline{A}\overline{C}=0$，$AC=0$。用卡诺图表示，图 10.22(a)中的 5 个 0 是 3 个限制填图的结果。

AB \ C	0	1
00	0	
01	0	
11		0
10	0	0

(a) 限制条件

AB \ C	0	1
00	0	1
01	0	1
11	1	0
10	0	0

(b) 完整卡诺图

图 10.22　用卡诺图求解 2-SAT 问题

图 10.22(b)中等于 1 的方格就是可行的答案，一共有 3 个 1：$\overline{A}\overline{B}C$、$\overline{A}BC$、$AB\overline{C}$。也就是 3 个合法出席方案：$A$ 女＋B 女＋C 男，A 女＋B 男＋C 男，A 男＋B 男＋C 女。

2-SAT 的可行解有多少个？在上面卡诺图的图解中可以发现，卡诺图的方格有 2^n 个，也就是说，可行解的数量是 $O(2^n)$的，复杂度很高，所以一般不会要求输出所有的解，只需要判断序列是否存在，或者只输出一个可行解。

2. 2-SAT 问题的定义

根据上面的例子，给出 SAT 问题的定义，它本身是一个数字逻辑问题：有 n 个布尔变量(布尔变量的特点是只有 0、1 两个值)，其中一些布尔变量之间有限制关系；用所有 n 个布尔变量组成序列，使得其满足所有限制关系；判断序列是否存在。这就是 SAT(Satisfiability)问题。如果每个限制关系只涉及两个变量，则是 2-SAT 问题。

3. 用图论的方法解决 2-SAT 问题

(1) 首先，把矛盾关系用图来表示。

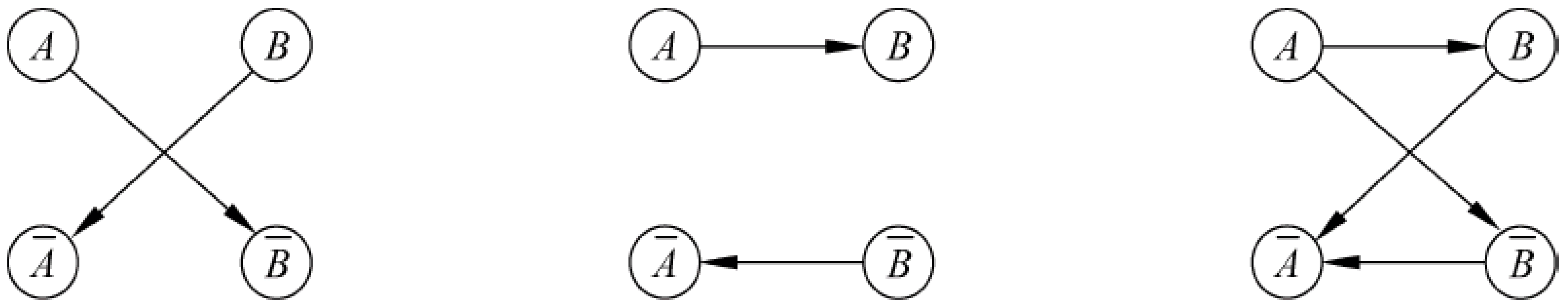
举一个简单例子。有两对夫妻 A、B，有两个限制：A、B 矛盾，A、$\overline{B}$ 矛盾。

先看 A、B 的矛盾，有两个推论：如果 A 确定出席，那么只能 $\overline{B}$ 出席，用 $A\rightarrow\overline{B}$ 表示，表

① 在数字逻辑中有 3 种基本逻辑操作，即与、或、非。非：$\overline{A}$ 是 A 的反变量。或：$A+\overline{A}=1$。与：$A\overline{A}=0$。

示"有 A 必有 $\overline{B}$"；如果 B 确定出席，只能 $\overline{A}$ 出席，用 $B \rightarrow \overline{A}$ 表示。

A、B 这一对矛盾，推出了两个结果，这是因为 A、B 是对等的，所以产生的关系是**对称**的。见下面的有向图 10.23(a)。



(a) A、B 有矛盾　(b) A、$\overline{B}$ 矛盾　(c) 合起来

图 10.23　用图表示 A、B 的矛盾关系

同样，A、$\overline{B}$ 矛盾，推论是 $A \rightarrow B$、$\overline{B} \rightarrow \overline{A}$，见有向图 10.23(b)(可以观察到，这里推论出 A、B 同时出席，和前一个限制正好矛盾)。

两个限制合起来的有向图是图 10.23(c)。这个有向图的点包含了所有人，有向边说明了依赖关系。

(2) 合法的出席组合和强连通分量 SCC 的关系。

在最后的图 10.23(c)中，形成了多个强连通分量 SCC。一个 SCC 内部的点都是互相依赖的，也就是说，如果有一个出席，那么这个 SCC 内部的所有人都要出席。所以，一个 SCC 内部不应该有夫妻关系，因为夫妻只能出席一人。只要所有的 SCC 内部都没有夫妻，就会有合法的出席组合。为深入理解这一点，读者可以观察图 10.23(c)，所有的点都不是强连通的，每个点都是独立的 SCC，所以这个图有合法的解。特别要注意其中有 $A \rightarrow B \rightarrow \overline{A}$，但 A 和 $\overline{A}$ 并不是强连通的。

所以，程序的步骤是根据给定的限制条件建图，计算 SCC，如果每个 SCC 内部都没有夫妻，就说明有合法的出席组合。

(3) 在图上求解一个合法组合。作为参照，读者可以先用上面卡诺图的方法得出有 $\overline{A}B$、$\overline{A}\overline{B}$ 两种出席组合。

读者可能觉得，只要在图 10.23(c)中沿着一条路径按顺序找，就能找到一个合法组合，因为一条路径上前后的点都是相互依赖的。但是，其实这个从前到后的顺序是不对的，应该按反序找，即从最后的点开始往前，这才是对的。这是因为最后的点是依赖性最大的，例如图 10.23(c)中的 $\overline{A}$，它被前面的 B 和 $\overline{B}$ 所依赖。

把每个 SCC 看成一个点，构成了一个 DAG 图，进行反图的拓扑排序，在选中点的时候同时排除图中相矛盾的点，就能找到合法的组合。

在编程时并不需要再做一次拓扑排序。在求 SCC 时已经得到了每个点所属的 SCC，SCC 的序号就是一个拓扑排序。

【习题】

hdu 3062 "Party"，2-SAT 简单题。

hdu 1824 "Let's go home"，简单题。

hdu 4115 "Eliminate the Conflict"。

hdu 4421 "Bit Magic"。

10.9 最 短 路

最短路径是图论中最为人们熟知的问题。

1. 最短路径问题

在一个图中有 n 个点、m 条边。边有权值，例如费用、长度等，权值可正可负。边可能是有向的，也可能是无向的。给定两个点，起点是 s，终点是 t，在所有能连接 s 和 t 的路径中寻找边的权值之和最小的路径，这就是最短路径问题。

2. 可加性参数和最小性参数

这两种参数区分了最短路径问题和网络流问题。

在最短路径问题中，是计算"路径上边的权值之和"。边的权值是"可加性参数"，例如费用、长度等，它们是"可加的"，一条路径上的总权值是这条路径上所有边的权值之和。下一节的"最小生成树"问题，边的权值也是"可加性参数"。

但是，在网络流问题中是找"路径上权值最小的边"。例如"最大流"问题，边的权值是"最小性参数"。比如水流，一条路径上的能流过的水流取决于这条路径上容量最小的那条边。再比如网络的带宽，一条网络路径上的整体带宽是这条路径上带宽最小的那条边的带宽。

3. 用 DFS 搜索所有的路径

在一般的图中，求图中任意两点间的最短路径，首先需要遍历所有可能经过的结点和边，不能有遗漏；其次，在所有可能的路径中查找最短的一条。如果用暴力法找所有路径，最简单的方法是把 n 个结点进行全排列，然后从中找到最短的。但是共有 $n!$ 个排列，是天文数字，无法求解。更好的办法是用 DFS 输出所有存在的路径，这显然比 $n!$ 要少得多，不过，其复杂度仍然是指数级的。

4. 用 BFS 求最短路径

在特殊的地图中，所有的边都是无权的，可以把每个边的权值都设成 1，那么 BFS 也是很好的最短路径算法，这些内容在 4.3.3 节中已经提到，请读者回顾有关内容。

下面讲解常见的 4 个最短路径算法。这几种方法差别很大，如果读者不能理解其思想，学起来容易头晕。为清晰地讲解这些算法，本书从 3 个方面展开：先结合现实中的模型讲解算法的思想；然后解释编程的逻辑过程；最后给出标准程序，这些程序结合了不同的数据结构和 STL 库。

最短路径的 4 个常用算法是 Floyd、Bellman-Ford、SPFA、Dijkstra。在不同的应用场景下，用户应该有选择地使用它们：

(1) 图的规模小，用 Floyd。如果边的权值有负数，需要判断负圈。

(2) 图的规模大，且边的权值非负，用 Dijkstra。

(3) 图的规模大，且边的权值有负数，用 SPFA。需要判断负圈。

再具体一点，可以总结出表 10.1。

表 10.1 对比 4 种常用算法

结点 n、边 m	边权值	选用算法	数据结构
$n<200$	允许有负	Floyd	邻接矩阵
$n\times m<10^7$	允许有负	Bellman-Ford	邻接表
更大	有负	SPFA	邻接表、前向星
	无负数	Dijkstra	邻接表、前向星

本节后面的讲解都以基础题 hdu 2544 为例，讲解不同算法的思想，并给出模板代码。

hdu 2544 "最短路径"

把衣服从商店运到赛场，寻找从商店到赛场的最短路径线。

有 N 个路口，标号为 1 的路口是商店所在地，标号为 N 的路口是赛场所在地。有 M 条路，每条路的数据包括 3 个整数 A、B、C，表示路口 A 与路口 B 之间有一条路，需要 C 分钟的时间走过这条路。

作为预习，读者可以尝试用 DFS 做这一题，暴力搜索出所有可能的路径。在编程时注意用剪枝技术进行优化，如果新路径搜到一半已经比以前得到的最短路径更长，就停止搜这个路径，重新开始搜下一个。

10.9.1 Floyd-Warshall

1. 所有点对间的最短路径

如何一次性求所有结点之间的最短距离？Floyd 可以完成这一工作，其他 3 种算法都不行。而且 Floyd 是最简单的最短路径算法，程序比暴力的 DFS 更简单。需要提醒的是，Floyd 的复杂度很高，只能用于小规模的图。

Floyd 用到了**动态规划**的思想：求两点 i、j 之间的最短距离，可以分两种情况考虑，即经过图中某个点 k 的路径和不经过点 k 的路径，取两者中的最短路径。

动态规划的过程可以描述为：

(1) 令 $k=1$，计算所有结点之间(经过结点 1、不经过结点 1)的最短路径。

(2) 令 $k=2$，计算所有结点之间(经过结点 2、不经过结点 2)的最短路径，这一次计算利用了 $k=1$ 时的计算结果。

(3) 令 $k=3$，……

读者可以这样想象这个过程：

(1) 图上有 n 个结点，m 条边。

(2) 把图上的每个点看成一个灯，初始时灯都是灭的，大部分结点之间的距离被初始化为无穷大 INF，除了 m 条边连接的那些结点以外。

(3) 从结点 $k=1$ 开始操作，想象点亮了这个灯，并以 $k=1$ 为中转点，计算和调整图上所有点之间的最短距离。很显然，对这个灯的邻居进行的计算是有效的，而对远离它的那些点的计算基本是无效的。

(4) 逐步点亮所有的灯，每次点灯，就用这个灯中转，重新计算和调整所有灯之间的最

短距离，这些计算用到了以前点灯时得到的计算结果。

(5) 灯逐渐点亮，直到图上的点全亮，计算结束。

在这个过程中，由于很多计算是无效的，所以算法的效率不高。

复杂度。在下面的程序中，函数 floyd()有 3 重循环，复杂度是 $O(n^3)$，只能用于计算规模很小的图，即 $n<200$ 的情况。

hdu 2544 的 Floyd 算法代码(邻接矩阵)

```
include<bits/stdc++.h>
using namespace std;
const int INF = 1e6;                        //路口之间的初始距离,看成无穷大,相当于断开
const int NUM = 105;
int graph[NUM][NUM];                        //邻接矩阵存图
int n, m;
void floyd() {
    int s = 1;                              //定义起点
    for(int k = 1; k <= n; k++)             //floyd()的 3 重循环
        for(int i = 1; i <= n; i++)
            if(graph[i][k]! = INF)          //一个小优化,在 hdu 1704 题中很必要
            for(int j = 1; j <= n; j++)     //请思考: 把 k 循环放到 i、j 之后行不行
                if(graph[i][j] > graph[i][k] + graph[k][j])
                   graph[i][j] = graph[i][k] + graph[k][j];
//graph[i][j] = min(graph[i][j], graph[i][k] + graph[k][j]);
//上面两句这样写也行,但是 min()比较慢,如果图大,可能会超时.读者可以试试 poj 3259
    printf("%d\n", graph[s][n]);            //输出结果
}
int main() {
    while(~scanf("%d%d", &n, &m)) {
                    //如果图的数据很大,不能用 cin 这种慢的输入
        if(n == 0 && m == 0)  return 0;
        for(int i = 1; i <= n; i++)         //邻接矩阵初始化
            for(int j = 1; j <= n; j++)
                graph[i][j] = INF;          //任意两点间的初始距离为无穷大
        while(m--) {
            int a, b, c;
            scanf("%d%d%d", &a, &b, &c);
            graph[a][b] = graph[b][a] = c;  //邻接矩阵存图
        }
        floyd();
    }
    return 0;
}
```

Floyd 算法虽然低效，但是也有优点：

(1) 程序很简单；

(2) 可以一次求出所有结点之间的最短路径；

(3) 能处理有负权边的图。

2. 判断负圈

程序中有一个有趣的地方。在程序中，结点 i 到自己的距离 graph[i][i]并没有置初值

为 0，而是 INF，读者可能觉得很奇怪；并且在计算结束之后，graph[i][i]也不是 0，而是 graph[i][i]=graph[i][u]+…+graph[v][i]，即到外面绕一圈回来的最小路径。这一点可用于判断**负圈**。

负圈是这样产生的：如果某些边的权值为负数，那么图中可能有这样的环路，环路上边的权值之和为负数，这样的环路就是负圈。每走一次这个负圈，总权值就会更小，导致陷在这个圈里出不来。

利用 Floyd 算法很容易判断负圈，只要在 floyd()中判断是否存在某个 graph[i][i]<0 就行了。因为 graph[i][i]是 i 到外面绕一圈回来的最小路径，如果小于 0，说明存在负圈。此时可置 graph[i][i]的初值为 0，这样能加快判断过程。请读者练习 poj 3259 "Wormholes"题。

3. Floyd 与邻接矩阵

上面的程序用邻接矩阵存图，实现 Floyd。邻接矩阵十分浪费空间，那么用邻接表是否会更好呢？答案是在 Floyd 算法中邻接表并不比邻接矩阵好，除非两点之间有多个边，导致不能用邻接矩阵表示。因为 Floyd 的计算过程是用动态规划求所有点之间的最短距离，必须用一个 $n \times n$ 的矩阵记录状态，空间无法节省。存图的邻接矩阵可以同时用来记录状态。

4. 打印路径

有时候题目需要打印路径，请读者练习 hdu 1385 "Minimum Transport Cost"。如果有疑问，可以先学习下面的几个算法，本书都给出了打印路径的方法。

【习题】

hdu 1599 “find the mincost route”，求最小环。

hdu 3631 “Shortest Path”，Floyd 变形。

hdu 1704 “rank”，需要在 floyd()中加一个优化：if(graph[i][k]!=INF)。

10.9.2 Bellman-Ford

1. Bellman-Ford 算法

Bellman-Ford[①] 算法用来解决单源最短路径问题：给定一个起点 s，求它到图中所有 n 个结点的最短路径。

Bellman-Ford 算法的特点是只对相邻结点进行计算，可以避免 Floyd 那种大撒网式的无效计算，大大提高了效率。为理解这个算法，可以想象图上的每个点都站着一个人，初始时，所有人到 s 的距离设为 INF，即无限大。用下面的步骤求最短路径：

(1) 第一轮，给所有的 n 个人每人一次机会，问他的邻居到 s 的最短距离是多少？如果他的邻居到 s 的距离不是 INF，他就能借道这个邻居到 s 去，并且把自己原来的 INF 更新为较短的距离。显然，开始的时候，起点 s 的直连邻居(例如 u)肯定能更新距离，而 u 的邻居(例如 v)，如果在 u 更新之后问 v，那么 v 有机会更新，否则就只能保持 INF 不变。特别地，在第一轮更新中，存在一个与 s 最近的邻居 t，t 到 s 的直连距离就是全图中 t 到 s 的最短距

① Bellman-Ford 的历史与改进：https://en.wikipedia.org/wiki/Bellman-Ford_algorithm(短网址：t.cn/RSrredV)。

离。因为它通过别的邻居绕路到 s，肯定更远。t 的最短距离已经得到，后面不会再更新。在很多教材中把一轮更新称为一次“**松弛(relax)**”，这个概念也用在 Dijkstra 算法中。

(2) 第二轮，重复第一轮的操作，再给每个人一次问邻居的机会。这一轮操作之后，至少存在一个 s 或 t 的邻居 v，可以算出它到 s 的最短距离。v 要么与 s 直连，要么是通过 t 到达 s 的。v 的最短距离也得到了，后面不会再更新。

(3) 第三轮，再给每个人一次机会……

继续以上操作，直到所有人都不能再更新最短距离为止。

一共需要几轮操作呢？每一轮操作都至少有一个新的结点得到了到 s 的最短路径。所以，最多只需要 n 轮操作就能完成 n 个结点。在每一轮操作中，需要检查所有 m 个边，更新最短距离。根据以上分析，**Bellman-Ford 算法的复杂度是 $O(nm)$**。

以上过程，每个结点可以独立进行计算，所以这个算法符合并行计算的思想，可以用在并行计算上。例如计算机网络的 BGP 路由协议，每个路由器是一个结点，它根据与邻居的信息交换，独自计算到网络中其他路由器的最短距离。BGP 是 Bellman-Ford 算法(更准确地说，是下面的 SPFA 算法)的一个典型应用。

Bellman-Ford 有现实的模型，即问路。每个十字路口站着一个警察；在某个路口，路人问一个警察，怎么走到 s 最近？如果这个警察不知道，他会问相邻几个路口的警察：“从你这个路口走，能到 s 吗？有多远？”这些警察可能也不知道，他们会继续问新的邻居。这样传递下去，最后肯定有个警察是 s 路口的警察，他会把 s 的信息返回给他的邻居，邻居再返回给邻居。最后所有的警察都知道怎么走到 s，而且是最短的路：从 s 返回信息到所有其他点的过程就像在一个平静的池塘中从 s 丢下一个石头，荡起的涟漪一圈圈向外扩散，这一圈圈涟漪经过的路径肯定是最短的。

视频讲解

问路模型里有趣的一点，并且能体现 Bellman-Ford 思想的是警察并不需要知道到 s 的完整的路径，他只需要知道从自己的路口出发往哪个方向走能到达 s，并且路最近。

下面是 hdu 2544 的 Bellman-Ford 程序，用 bellman()替换上一节的 floyd()即可。

hdu 2544 的 Bellman-Ford 算法代码(邻接矩阵)

```
void bellman(){
    int s = 1;                          //定义起点
    int d[NUM];                         //d[i]记录结点 i 到起点 s 的最短距离.本题 s = 1
    for(int i = 1; i <= n; i++)
        d[i] = INF;                     //所有结点到 s 的距离初始化为无穷大
    d[s] = 0;                           //以上是初始化 d[]
    for(int k = 1; k <= n; k++)         //n 轮操作
        for(int i = 1; i <= n; i++)
            //i 和 j: 处理图中存在的边,即 graph[i][j]不等于 INF 的边
        for(int j = 1; j <= n; j++)
          if(d[j] > d[i] + graph[i][j])
                    //j 通过 i 到达起点 s: 如果距离更短,更新
            d[j] = d[i] + graph[i][j];
    printf("%d\n",d[n]);                //输出结果
}
```

但是上面的代码并没有实用价值。由于使用了邻接矩阵这种不合适的数据结构，它没有发挥出 Bellman-Ford 的威力。Bellman-Ford 的每一轮操作只需要检查存在的 m 条边。在 $n \times n$ 的邻接矩阵中，这 m 条边是那些不等于 INF 的边，但是上面的程序却不得不检查所有 $n \times n$ 条边。

下面的程序对存储进行了优化，用 struct edge e[10005]数组来存 m 条边，避免了存储那些不存在的边。这种简单的存储方法不是邻接表，不能快速搜一个结点的所有邻居，不过正适合 Bellman-Ford 这种简单的算法。

hdu 2544 的 Bellman-Ford 算法代码（数组存边）

```
include <bits/stdc++.h>
using namespace std;
const int INF = 1e6;
const int NUM = 105;
struct edge { int u, v, w; } e[10005];            //边：起点 u,终点 v,权值 w
int n, m, cnt;
int pre[NUM];
    //记录前驱结点.pre[x] = y,在最短路径上,结点 x 的前一个结点是 y
void print_path(int s, int t) {                   //打印从 s 到 t 的最短路径
   if(s == t){ printf("%d ", s); return; }        //打印起点
   print_path(s, pre[t]);                         //先打印前一个点
   printf("%d ", t);                              //后打印当前点.最后打印的是终点 t
}
void bellman(){
    int s = 1;                                    //定义起点
    int d[NUM];                                   //d[i]记录第 i 个结点到起点 s 的最短距离
    for (int i = 1; i <= n; i++)     d[i] = INF;  //初始化为无穷大
    d[s] = 0;
    for (int k = 1; k <= n; k++)                  //一共有 n 轮操作
        for (int i = 0; i < cnt; i++){            //检查每条边
            int x = e[i].u,  y = e[i].v;
            if (d[x] > d[y] + e[i].w){
                    //x 通过 y 到达起点 s：如果距离更短,更新
                d[x] = d[y] + e[i].w;
                pre[x] = y;                       //如果有需要,记录路径
            }
        }
        printf("%d\n", d[n]);
        //print_path(s,n);                        //如果有需要,打印路径
}
int main() {
    while(~scanf("%d%d", &n, &m)) {
        if(n == 0 && m == 0) return 0;
        cnt = 0;                                  //记录边的数量.本题的边是双向的,共有 2m 条
        while (m--) {
            int a,b,c;
            scanf("%d%d%d",&a,&b,&c);
            e[cnt].u = a;  e[cnt].v = b;  e[cnt].w = c;  cnt++;
            e[cnt].u = b;  e[cnt].v = a;  e[cnt].w = c;  cnt++;
```

```
        }
        bellman();
    }
    return 0;
}
```

2. 打印最短路径

计算出最短距离后，如果要打印整个路径，十分容易。

对于单源最短路径算法 Bellman-Ford（以及后面讲到的 Dijkstra），在连通图中，从起点 s 到任意一个结点 t 都有一条最短路径（如果有多条最短路径，就简单地选其中一条，其他的丢弃）；反过来看，从任意一个结点 t 往前追溯，沿着最短路径，一个结点一个结点往回走，就能到达起点 s。所以，只要在每个结点上记录它的前驱结点就行了。

定义 pre[]记录前驱结点。pre[x]$=y$ 的意思是在最短路径上结点 x 的前一个结点是 y。然后用 print_path()打印整个路径。

3. 判断负圈

Bellman-Ford 也能判断负圈。当没有负圈时，只需要 n 轮就结束。如果超过 n 轮，最短路径还有变化，那么肯定有负圈。

判断负圈的程序可以写在两个 for 循环结束后。检查所有的边，如果存在某个边 (u,v)，有 $d(u)>d(v)+w(u,v)$，说明 $d(u)$的更新未结束，还能更新为更小的值。这只能是负圈引起的。

更紧凑的程序可以这样写：在循环内部判断是不是超过了 n 轮。程序如下：

hdu 2544 的 Bellman-Ford 算法代码（有判断负圈的功能）

```
void bellman(){
    int d[NUM];
    for (int i = 2;i <= n;i++)
       d[i] = INF;
    d[1] = 0;
    int k = 0;                                  //记录有几轮操作
    bool update = true;                         //判断是否有更新
    while(update) {
       k++;
       update = false;
       if(k > n) {printf("有负圈"); return;}     //有负圈,停止
       for (int i = 0; i < cnt; i++){
           int x = e[i].u, y = e[i].v;
           if (d[x] > d[y] + e[i].w){
              update = true;
              d[x] = d[y] + e[i].w;
           }
       }
    }
    printf("%d\n",d[n]);
}
```

10.9.3 SPFA

用队列处理 Bellman-Ford 算法可以很好地优化，这种方法叫作 SPFA。SPFA 的效率很高，在算法竞赛中的应用很广泛。

Bellman-Ford 算法有很多低效或无效的操作。分析 Bellman-Ford 算法，其核心部分是在每一轮操作中更新**所有**结点到起点 s 的最短距离。根据前面的讨论可知，计算和调整一个结点 u 到 s 的最短距离后，如果紧接着调整 u 的**邻居**结点，这些邻居肯定有新的计算结果；而如果漫无目的地计算不与 u 相邻的结点，很可能毫无变化，所以这些操作是低效的。

因此，在计算结点 u 之后，下一步只计算和调整它的邻居，这样能加快收敛的过程。这些步骤可以用队列进行操作，这就是 SPFA。

SPFA 很像 BFS：

(1) 起点 s 入队，计算它所有邻居到 s 的最短距离(当前最短距离，不是全局最短距离。在下文中，把计算一个结点到起点 s 的最短路径简称为更新状态。最后的“状态”就是 SPFA 的计算结果)。把 s 出队，状态有更新的邻居入队，没更新的不入队。也就是说，队列中都是状态有变化的结点，只有这些结点才会影响最短路径的计算。

(2) 现在队列的头部是 s 的一个邻居 u。弹出 u，更新其所有邻居的状态，把其中有状态变化的邻居入队列。

(3) 这里有一个问题，弹出 u 之后，在后面的计算中 u 可能会再次更新状态(后来发现，u 借道其他结点去 s，路更近)。所以，u 可能需要重新入队列。这一点很容易做到：在处理一个新的结点 v 时，它的邻居可能就是以前处理过的 u，如果 u 的状态变化了，把 u 重新加入队列就行了。

(4) 继续以上过程，直到队列空。这也意味着所有结点的状态都不再更新。最后的状态就是到起点 s 的最短路径。

上面第(3)点决定了 SPFA 的效率。有可能只有很少结点重新进入队列，也有可能很多。这取决于图的特征，即使两个图的结点和边的数量一样，但是边的权值不同，它们的 SPFA 队列也可能差别很大。所以，**SPFA 是不稳定的**。

在比赛时，有的题目可能故意卡 SPFA 的不稳定性：如果一个题目的规模很大，并且边的权值为非负数，它很可能故意设置了不利于 SPFA 的测试数据。此时不能冒险用 SPFA，而是用下一节的 Dijkstra 算法。Dijkstra 是一种稳定的算法，一次迭代至少能找到一个结点到 s 的最短路径，最多只需要 m(边数)次迭代即可完成。

1. 基于邻接表的 SPFA

在这个程序中，存图最合适的方法是邻接表。上面第(2)步是更新 u 的所有邻居结点的状态，而邻接表可以很快地检索一个结点的所有邻居，正符合算法的需要。程序 main()输入图时，每执行一次 e[a]. push_back(edge(a,b,c))，就把边(a,b)存到了结点 a 的邻接表中；在 spfa()中，执行 for(int i=0; i<e[u]. size(); i++)，就检索了结点 u 的所有邻居。

hdu 2544 的 SPFA 算法代码(邻接表+队列)

```
include <bits/stdc++.h>
```

```
using namespace std;
const int INF = 1e6;
const int NUM = 105;
struct edge{
    int from, to, w;
//边：起点 from,终点 to,权值 w.from 并没有用到,e[i]的 i 就是 from
    edge(int a, int b,int c){from = a; to = b; w = c;}
};
vector < edge > e[NUM];                    //e[i]：存第 i 个结点连接的所有边
int n, m;
int pre[NUM];
 //记录前驱结点.pre[x] = y,在最短路径上,结点 x 的前一个结点是 y
void print_path(int s,int t) {             //打印从 s 到 t 的最短路径
    ;                               //内容与 Bellman - Ford 程序中的 print_path()完全一样
}
int spfa(int s){
    int dis[NUM];                          //记录所有结点到起点的距离
    bool inq[NUM];                         //inq[i] = true 表示结点 i 在队列中
    int Neg[NUM];                          //判断负圈(Negative loop)
    memset(Neg, 0, sizeof(Neg));
    Neg[s] = 1;
    for(int i = 1;i <= n;i++) { dis[i] = INF;  inq[i] = false; }     //初始化
    dis[s] = 0;                            //起点到自己的距离是 0
    queue < int > Q;
    Q.push(s);
    inq[s] = true;                         //起点进队列
    while(!Q.empty()) {
        int u = Q.front();
        Q.pop();                           //队头出队
        inq[u] = false;
        for(int i = 0; i < e[u].size(); i++) {    //检查结点 u 的所有邻居
            int v = e[u][i].to, w = e[u][i].w;
            if (dis[u] + w < dis[v])  {
                  //u 的第 i 个邻居 v,它借道 u,到 s 更近
              dis[v] = dis[u] + w;         //更新第 i 个邻居到 s 的距离
              pre[v] = u;                  //如果有需要,记录路径
              if(!inq[v]) {
        //第 i 个邻居更新状态了,但是它不在队列中,把它放进队列
                    inq[v] = true;
                    Q.push(v);
                    Neg[v]++;
                    if(Neg[v] > n) return 1;     //出现负圈
                  }
              }
          }
     }
     printf("%d\n",dis[n]);
     //print_path(s,n);                    //如果有需要,打印路径
     return 0;
}
int main(){
```

```
    while(~scanf("%d%d",&n,&m)) {
        if(n == 0 && m == 0)      return 0;
        for(int i = 1; i <= n; i++)    e[i].clear();
        while(m -- ) {
            int a,b,c;
            scanf("%d%d%d", &a, &b, &c);
            e[a].push_back(edge(a,b,c));
                    //结点 a 的邻居,都放在 node[a]里
            e[b].push_back(edge(b,a,c));
        }
        spfa(1);                                //起点是 1
    }
    return 0;
}
```

前面在讲 Bellman-Ford 的时候,曾提到它适合并行计算。读者可以发现,SPFA 比 Bellman-Ford 能更有效率地进行并行计算。例如前面提到的问路的例子,每个警察只需要在某个邻居警察通知有路径变化之后才进行计算,并把变化传递给别的邻居;如果没有收到邻居发来的变化信息,警察不需要做任何动作。这正是 SPFA 的思想。

判断负圈。SPFA 也适用于有负权值的图,也能判断负圈。如果有一个点进队列超过 n 次,那就说明图中存在负圈。具体见程序中与 Neg[]有关的部分。

打印最短路径。和前面 Bellman-Ford 打印最短路径非常相似。定义 pre[]记录前驱结点,然后用 print_path()打印整个路径。具体内容见程序。

2. 基于链式前向星的 SPFA

上面的基于邻接表的代码已经很好了,不过,在极端的情况下,图特别大,用邻接表也会超空间限制,此时就需要用到前面提到的链式前向星来存图。

建议读者认真消化下面的代码,内容包括链式前向星存图、SPFA 算法、打印最短距离、打印路径、判断负圈。这是本书精心整理的一套模板。

读者可以套用这个模板,试试 hdu 1535 "Invitation Cards"题。hdu 1535 题的图有 100 万个点,如果不用链式前向星,用别的数据结构很容易发生 MLE 错误。

hdu 2544 的 SPFA 算法代码(链式前向星)

```
include <bits/stdc++.h>
using namespace std;
const int INF = INT_MAX / 10;
const int NUM = 1000005;            //一百万个点,一百万个边
struct Edge{                        //边: edge[i]的 i 就是起点,终点 to,权值 w.下一个边 next
    int to, next, w;
}edge[NUM];
int n, m, cnt;
int head[NUM];
int dis[NUM];                       //记录所有结点到起点的距离
bool inq[NUM];                      //inq[i] = true 表示结点 i 在队列中
int Neg[NUM];                       //判断负圈(Negative loop)
int pre[NUM];                       //记录前驱结点
```

```
void print_path(int s,int t) {          //打印从 s 到 t 的最短路径
    ;                                   //内容与 Bellman - Ford 程序中的 print_path()完全一样
}
void init(){
    for(int i = 0; i < NUM; ++i){
        edge[i].next = -1;
        head[i] = -1;
    }
    cnt = 0;
}
void addedge(int u, int v, int w){      //前向星存图
    edge[cnt].to = v;
    edge[cnt].w = w;
    edge[cnt].next = head[u];
    head[u] = cnt++;
}
int spfa(int s) {
    memset(Neg, 0, sizeof(Neg));
    Neg[s] = 1;
    for(int i = 1; i <= n; i++) { dis[i] = INF;  inq[i] = false; }//初始化
    dis[s] = 0;                                       //起点到自己的距离是 0
    queue<int> Q;
    Q.push(s);
    inq[s] = true;                                    //起点进队列

    while(!Q.empty()) {
        int u = Q.front(); Q.pop();                   //队头出队
        inq[u] = false;
        for(int i = head[u]; ~i; i = edge[i].next) {  //~i 也可以写成 i!= -1
            int v = edge[i].to, w = edge[i].w;
            if (dis[u] + w < dis[v]) {
                                //u 的第 i 个邻居 v,它借道 u,到 s 更近
                dis[v] = dis[u] + w;                  //更新第 i 个邻居到 s 的距离
                pre[v] = u;                           //如果有需要,记录路径
                if(!inq[v]) {
                  //邻居 v 更新状态了,但是它不在队列中,把它放进队列
                  inq[v] = true;
                  Q.push(v);
                  Neg[v]++;
                  if(Neg[v] > n) return 1;            //出现负圈
                }
            }
        }
    }
    printf("%d\n",dis[n]);                            //从 s 到 n 的最短距离
    //print_path(s,n);                                //如果有需要,打印路径
    return 0;
}
int main() {
    while(~scanf("%d%d",&n,&m)) {
        init();
```

```
        if(n == 0 && m == 0) return 0;
        while(m -- ) {
          int u,v,w;
          scanf("%d%d%d", &u, &v, &w);
          addedge(u,v,w);
          addedge(v,u,w);
        }
        spfa(1);
    }
    return 0;
}
```

10.9.4 Dijkstra

Dijkstra 算法也用来解决单源最短路径问题。Dijkstra 是非常高效而且稳定的算法，它比前面提到的最短路径算法都复杂一些，下面先介绍它的思想。

前面在讲 Bellman-Ford 算法时，提到它在现实中的模型是找警察问路。在现实中，Dijkstra 有另外的模型，例如多米诺骨牌，读者可以想象下面的场景。

在图中所有的边上排满多米诺骨牌，相当于把骨牌看成图的边。一条边上的多米诺骨牌数量和边的权值(例如长度或费用)成正比。规定所有骨牌倒下的速度都是一样的。如果在一个结点上推倒骨牌，会导致这个结点上的所有骨牌都往后面倒下去。

在起点 s 推倒骨牌，可以观察到，从 s 开始，它连接的边上的骨牌都逐渐倒下，并到达所有能达到的结点。在某个结点 t，可能先后从不同的线路倒骨牌过来；先倒过来的骨牌，其经过的路径肯定就是从 s 到达 t 的最短路径；后倒过来的骨牌，对确定结点 t 的最短路径没有贡献，不用管它。

从整体看，这就是一个从起点 s 扩散到整个图的过程。

在这个过程中，观察所有结点的最短路径是这样得到的：

(1) 在 s 的所有直连邻居中，最近的邻居 u，骨牌首先到达。u 是第一个确定最短路径的结点。从 u 直连到 s 的路径肯定是最短的，因为如果 u 绕道别的结点到 s，必然更远。

(2) 然后，把后面骨牌的倒下分成两个部分，一部分是从 s 继续倒下到 s 的其他的直连邻居，另一部分是从 u 出发倒下到 u 的直连邻居。那么下一个到达的结点 v 必然是 s 或者 u 的一个直连邻居。v 是第二个确定最短路径的结点。

(3) 继续以上步骤，在每一次迭代过程中都能确定一个结点的最短路径。

Dijkstra 算法应用了**贪心法**的思想，即"抄近路走，肯定能找到最短路径"。

在上述步骤中可以发现：Dijkstra 的每次迭代，只需要检查上次已经确定最短路径的那些结点的邻居，检查范围很小，算法是**高效**的；每次迭代，都能得到至少一个结点的最短路径，算法是**稳定**的。

与 Bellman-Ford 对比：Bellman-Ford 是分布式的思想；而 Dijkstra 必须从起点 s 开始扩散和计算，是集中式的思想。读者可以试试在多米诺骨牌模型中运用 Bellman-Ford，看看行不行。

那么如何编程实现呢？程序的主要内容是维护两个集合，即已确定最短路径的结点集

合 A、这些结点向外扩散的邻居结点集合 B。程序逻辑如下：

(1) 把起点 s 放到 A 中，把 s 所有的邻居放到 B 中。此时，邻居到 s 的距离就是直连距离。

(2) 从 B 中找出距离起点 s 最短的结点 u，放到 A 中。

(3) 把 u 所有的新邻居放到 B 中。显然，u 的每一条边都连接了一个邻居，每个新邻居都要加进去。其中 u 的一个新邻居 v，它到 s 的距离 $\text{dis}(s,v)$ 等于 $\text{dis}(s,u)+\text{dis}(u,v)$。

(4) 重复(2)、(3)，直到 B 为空时结束。

计算结束后，可以得到从起点 s 到其他所有点的最短距离。

下面举例说明，如图 10.24 所示。

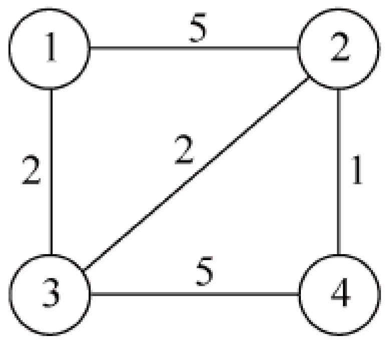


图 10.24　无向图

在图 10.24 中，起点是 1，求 1 到其他所有结点的最短路径。

(1) 1 到自己的距离最短，把 1 放到集合 A 里：$A=\{1\}$。把 1 的邻居放到集合 B 里：$B=\{(2-5),(3-2)\}$。其中(2—5)表示结点 2 到起点的距离是 5。

(2) 从 B 中找到离集合 A 最近的结点，是结点 3。在 A 中加上 3，现在 $A=\{1,3\}$，也就是说得到了从 1 到 3 的最短距离；从 B 中拿走(3—2)，现在 $B=\{(2-5)\}$。

(3) 对结点 3 的每条边，扩展它的新邻居，放到 B 中。3 的新邻居是 2 和 4，那么 $B=\{(2-5),(2-4),(4-7)\}$。其中(2—4)是指新邻居 2 通过 3 到起点 1，距离是 4。由于(2—4)比(2—5)更好，丢弃(2—5)，$B=\{(2-4),(4-7)\}$。

(4) 重复步骤(2)、(3)。从 B 中找到离起点最近的结点，是结点 2。在 A 中加上 2，并从 B 中拿走(2—4)；扩展 2 的邻居放到 B 中。现在 $A=\{1,3,2\}$，$B=\{(4-7),(4-5)\}$。由于(4—5)比(4—7)更好，丢弃(4—7)，$B=\{(4-5)\}$。

(5) 从 B 中找到离起点最近的结点，是结点 4。在 A 中加上 4，并从 B 中拿走(4—5)。此时已经没有新邻居可以扩展。现在 $A=\{1,3,2,4\}$，B 为空，结束。

下面讨论上述步骤的复杂度。图的边共有 m 个，需要往集合 B 中扩展 m 次。在每次扩展后，需要找集合 B 中距离起点最小的结点。集合 B 最多可能有 n 个结点。把问题抽象为每次往集合 B 中放一个数据，在 B 中的 n 个数中找最小值，如何快速完成？如果往 B 中放数据是乱放，找最小值也是用类似冒泡的简单方法，复杂度是 n，那么总复杂度是 $O(nm)$，和 Bellman-Ford 一样。

上述方法可以改进，得到更好的复杂度。改进的方法如下：

(1) 每次往 B 中放新数据时按从小到大的顺序放，用二分法的思路，复杂度是 $O(\log_2 n)$，保证最小的数总在最前面。

(2) 找最小值，直接取 B 的第一个数，复杂度是 $O(1)$。

此时 Dijkstra 算法总的复杂度是 $O(m\log_2 n)$，是最高效的最短路径算法。

在编程时，一般不用自己写上面的程序，直接用 STL 的优先队列就行了，完成数据的插入和提取。

下面的程序代码中有两个关键技术：

(1) 用邻接表存图和查找邻居。对邻居的查找和扩展是通过动态数组 vector < edge > e[NUM]实现的邻接表，和上一节的 SPFA 一样。其中 $e[i]$存储第 i 个结点上所有的边，边的一头是它的邻居，即 struct edge 的参数 to。在需要扩展结点 i 的邻居的时候，查找 $e[i]$即可。

已经放到集合 A 中的结点不要扩展；程序中用 bool done[NUM]记录集合 A，当 done[i]=true 时，表示它在集合 A 中，已经找到了最短路径。

(2) 在集合 B 中找距离起点最短的结点。直接用 STL 的优先队列实现，在程序中是 priority_queue <s_node> Q。但是有关丢弃的动作，STL 的优先队列无法做到。例如步骤(3)中，需要在 B={(2—5)，(2—4)，(4—7)}中丢弃(2—5)，但是 STL 没有这种操作。在程序中也是用 bool done[NUM]协助解决这个问题。从优先队列 pop 出(2—4)时，记录 done[2]=true，表示结点 2 已经处理好。下次从优先队列 pop 出(2—5)时，判断 done[2]是 true，丢弃。

下面是模板代码。

hdu 2544 的 Dijkstra 算法代码(邻接表+优先队列)

```
include <bits/stdc++.h>
using namespace std;
const int INF = 1e6;
const int NUM = 105;
struct edge{
    int from, to, w;
//边：起点，终点，权值.起点 from 并没有用到，e[i]的 i 就是 from
    edge(int a, int b,int c){from = a; to = b; w = c;}
};
vector <edge> e[NUM];                //用于存储图
struct s_node{
    int id, n_dis;                   //id：结点；n_dis：这个结点到起点的距离
    s_node(int b,int c){id = b; n_dis = c;}
    bool operator < (const s_node & a) const
    { return n_dis > a.n_dis;}
};
int n,m;
int pre[NUM];                        //记录前驱结点
void print_path(int s,int t)  {      //打印从 s 到 t 的最短路径
    ;                                //内容与 Bellman-Ford 程序中的 print_path()完全一样
}
void dijkstra(){
    int s = 1;                       //起点 s 是 1
    int  dis[NUM];                   //记录所有结点到起点的距离
    bool done[NUM];                  //done[i] = true 表示到结点 i 的最短路径已经找到
    for (int i = 1;i <= n;i++) {dis[i] = INF; done[i] = false; }  //初始化
    dis[s] = 0;                      //起点到自己的距离是 0
    priority_queue <s_node> Q;       //优先队列，存结点信息
    Q.push(s_node(s, dis[s]));       //起点进队列
    while (!Q.empty())  {
        s_node u = Q.top();          //pop 出距起点 s 距离最小的结点 u
        Q.pop();
        if(done[u.id])
            //丢弃已经找到最短路径的结点，即集合 A 中的结点
          continue;
        done[u.id] = true;
        for (int i = 0; i < e[u.id].size(); i++) {     //检查结点 u 的所有邻居
```

```
                edge y = e[u.id][i];                    //u.id 的第 i 个邻居是 y.to
                if(done[y.to])                          //丢弃已经找到最短路径的邻居结点
                    continue;
                if (dis[y.to] > y.w + u.n_dis) {
                    dis[y.to] = y.w + u.n_dis;
                    Q.push(s_node(y.to, dis[y.to]));
                                                        //扩展新的邻居,放到优先队列中
                    pre[y.to] = u.id;                   //如果有需要,记录路径
                }
            }
        }
        printf("%d\n", dis[n]);
        //print_path(s,n);                              //如果有需要,打印路径
    }
    int main(){
        while(~scanf("%d%d",&n,&m)) {
            if(n == 0 && m == 0) return 0;
            for (int i = 1;i <= n;i++)
                e[i].clear();
            while (m-- ) {
                    int a,b,c;
                scanf("%d%d%d",&a,&b,&c);
                e[a].push_back(edge(a,b,c));
                        //结点 a 的邻居,都放在 node[a]里
                e[b].push_back(edge(b,a,c));
            }
            dijkstra();
        }
    }
```

打印最短路径。和前面的 Bellman-Ford 算法一样,Dijkstra 打印最短路径也非常容易,原理和 Bellman-Ford 完全一样。首先定义 pre[]记录前驱结点,然后用 print_path()打印整个路径。具体内容见程序。

链式前向星。当图十分巨大时,需要用链式前向星存图。请读者自己总结模板,并做 hdu 1535 题。

【习题】

最短路径的题目很多,下面列出了一些训练题。

poj 1860/3259/1062/3037/3615/1511/3159(把差分约束转换为最短路径)。

hdu 1874/1596/2433/2680/4889/4568(最短路径+状态压缩 DP)。

10.10 最小生成树

最小生成树是无向图中的一个问题,也很常见。

在无向图中,连通而且不含有圈(环路)的图称为树。最小生成树(Minimal Spanning Tree,MST)的基本模型可以用下面的题目描述:

hdu 1233 "还是畅通工程"

有 n 个村庄需要修通道路，已知每两个村庄之间的距离，问怎么修路，使得所有村庄都连通(但不一定有直接的公路相连，只要能间接通过公路到达即可)，并且道路总长度最小？请计算最小的公路总长度。

图的两个基本元素是点和边，与此对应，有两种方法可以构造最小生成树 T。这两种算法都基于贪心法，因为 MST 问题满足贪心法的"最优性原理"，即全局最优包含局部最优。prim 算法的原理是"最近的邻居一定在 MST 上"，kruskal 算法的原理是"最短的边一定在 MST 上"。

(1) prim 算法：对点进行贪心操作。从任意一个点 u 开始，把距离它最近的点 v 加入到 T 中；下一步，把距离 $\{u,v\}$ 最近的点 w 加入到 T 中；继续这个过程，直到所有点都在 T 中。

(2) kruskal 算法：对边进行贪心操作。从最短的边开始，把它加入到 T 中；在剩下的边中找最短的边，加入到 T 中；继续这个过程，直到所有边都在 T 中。

在这两个算法中，重要的问题是判断圈。最小生成树显然不应该有圈，否则就不是"最小"了。所以，在新加入一个点或者边的时候要同时判断是否形成了圈。

10.10.1 prim 算法

图 10.25 说明了 prim 算法的步骤。设最小生成树中的点的集合是 U，开始时最小生成树为空，所以 U 为空。

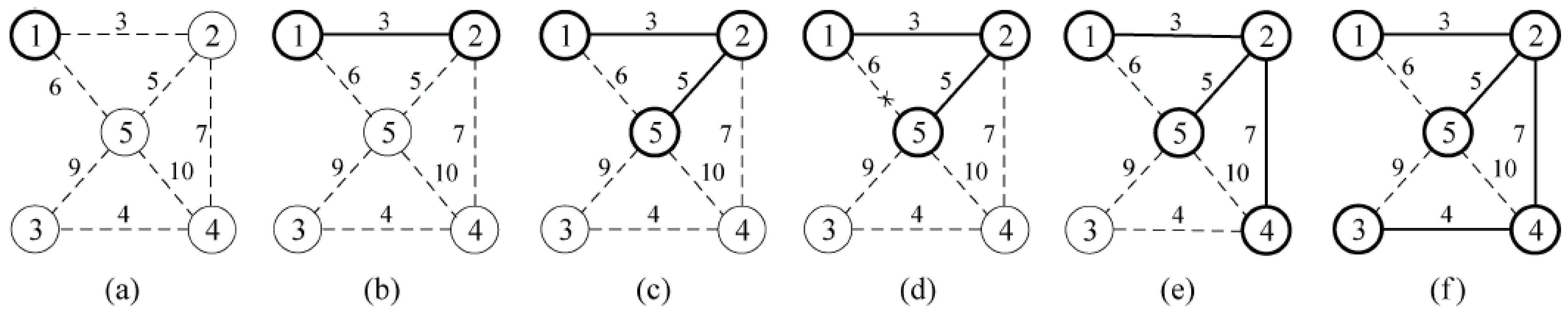


图 10.25　prim 算法

(1) 任取一点，例如点 1，放到 U 中，$U=\{1\}$，见图 10.25(a)。

(2) 找离集合 U 中的点最近的邻居，即 1 的邻居，是 2，放到 U 中，$U=\{1,2\}$，见图 10.25(b)。

(3) 找离 U 最近的点，是 5，$U=\{1,2,5\}$，见图 10.25(c)。

(4) 与 U 距离最短的是 1、5 之间的边，但是它没扩展新的点，不符合要求，见图 10.25(d)。

(5) 加入 4，$U=\{1,2,5,4\}$，见图 10.25(e)。

(6) 加入 3，$U=\{1,2,5,4,3\}$。所有点都在 U 中，结束，见图 10.25(f)。

上面的步骤和 Dijkstra 算法的步骤非常相似，不同的是 Dijkstra 需要更新 U 的所有邻居到起点的距离，即"松弛"，而 prim 不需要。所以，只要把 Dijkstra 的程序简化一些即可。

和 Dijkstra 一样，prim 程序如果用优先队列来查找距离 U 最近的点，能优化算法，此时复杂度是 $O(E\log_2 V)$。

prim 的编程比较麻烦，下面的 kruskal 算法是一种既简单又高效的算法。

10.10.2　kruskal 算法

kruskal 算法编程有以下两个关键技术：

(1) 对边进行排序。可以用 STL 的 sort()函数，排序后，依次把最短的边加入到 T 中。

(2) 判断圈，即处理连通性问题。这个问题用并查集简单而高效，并查集是 kruskal 算法的**绝配**。

仍以上面的图为例说明 kruskal 算法的操作步骤，如图 10.26 所示。

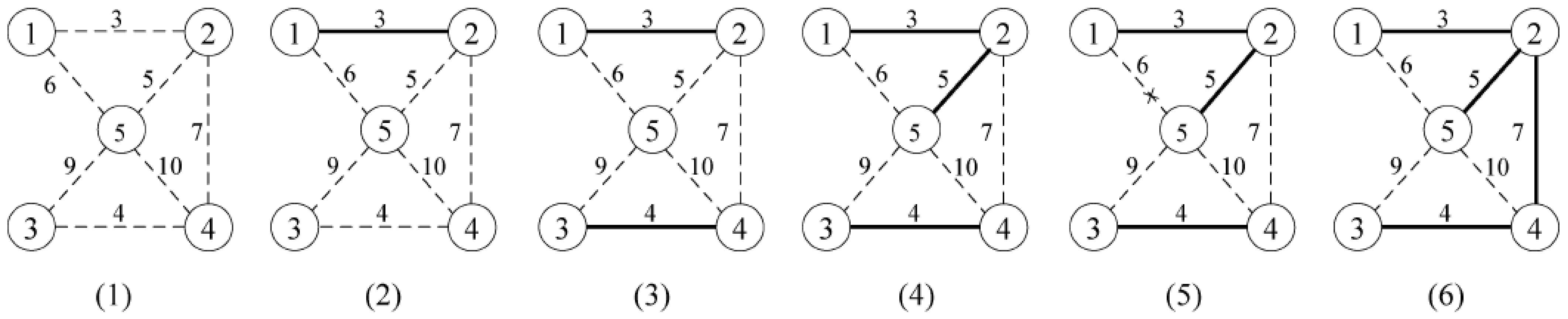


图 10.26　kruskal 算法

(1) 初始时最小生成树 T 为空，见图 10.26(1)。令 S 是以结点 i 为元素的并查集，在开始的时候，每个点属于独立的集(为了便于讲解，下表中区分了结点 i 和集 S，把集的编号加上了下画线)：

S	1	2	3	4	5
i	1	2	3	4	5

(2) 加入第一个最短边(1-2)：T={1-2}，见图 10.26(2)。在并查集 S 中，把结点 2 合并到结点 1，也就是把结点 2 的集2 改成结点 1 的集1。

S	1	***1***	3	4	5
i	1	2	3	4	5

(3) 加入第二个最短边(3-4)：T={1-2，3-4}，见图 10.26(3)。在并查集 S 中，结点 4 合并到结点 3。

S	1	1	3	***3***	5
i	1	2	3	4	5

(4) 加入第三个最短边(2-5)：T={1-2，3-4，2-5}，见图 10.26(4)。在并查集 S 中，把结点 5 合并到结点 2，也就是把结点 5 的集5 改成结点 2 的集1。在集1 中，所有结点都指向了根结点，这样做能避免并查集的**长链**问题。具体原理见 5.1 节的"路径压缩"的讲解。

S	1	1	3	3	***1***
i	1	2	3	4	5

(5) 第四个最短边(1-5)，见图 10.26(5)。检查并查集 S，发现 5 已经属于集1，丢弃这个边。这一步实际上是发现了一个**圈**。并查集的作用就体现在这里。

(6) 加入第五个最短边(2-4)，见图 10.26(6)。在并查集 S 中，把结点 4 的集并到结点 2 的集。注意这里结点 4 原来属于集3，实际上的修改是把结点 3 的集3 改成1。

S	1	1	***1***	3	1
i	1	2	3	4	5

(7) 对所有边执行上述操作，直到结束。读者可以练习加最后两个边(3-5)、(4-5)，这两个边都会形成圈。

下面是 hdu 1233 题的程序。

hdu 1233 题代码：kruskal+并查集

```
#include <bits/stdc++.h>
using namespace std;
const int NUM = 103;
int S[NUM];                                          //并查集
struct Edge {int u, v, w;} edge[NUM * NUM];          //定义边
bool cmp(Edge a, Edge b)  { return a.w < b.w; }
int find(int u) { return S[u] == u ? u : find(S[u]); }
              //查询并查集，返回 u 的根结点
int n, m;                                            //点，边
int kruskal() {
      int ans = 0;
      for(int i=1; i<=n; i++)
          S[i]=i;                                    //初始化，开始时每个村庄都是单独的集
      sort(edge+1, edge+1+m, cmp);
      for(int i = 1; i<=m; i++) {
         int b = find(edge[i].u);                    //边的前端点 u 属于哪个集
         int c = find(edge[i].v);                    //边的后端点 v 属于哪个集
         if(b == c) continue;                        //产生了圈，丢弃这个边
         S[c] = b;                                   //合并
         ans += edge[i].w;                           //计算 MST
      }
      return ans;
}
int main() {
    while(scanf("%d", &n), n) {
        m = n*(n-1)/2;
        for(int i = 1; i<=m; i++)                    //在题目中，点的编号从 1 开始
            scanf("%d%d%d",&edge[i].u, &edge[i].v, &edge[i].w);
        printf("%d\n", kruskal());
    }
    return 0;
}
```

kruskal 算法的复杂度包括两部分，即对边的排序 $O(E\log_2 E)$、并查集的操作 $O(E)$，一共是 $O(E\log_2 E+E)$，约等于 $O(E\log_2 E)$，时间主要花在排序上。

与 prim 相比，kruskal 的编码更简单，复杂度也好，更受人们欢迎。不过，如果图的边很多，kruskal 的复杂度要差一些。简单地说，kruskal 适用于稀疏图，prim 适用于稠密图。

【习题】

最小生成树算法有一些扩展问题，例如最大生成树、次小生成树、最小瓶颈生成树等，见下面的习题。

hdu 1102，简单题。

hdu 3938，离线算法。

poj 2377，最大生成树。

hdu 5627，最大生成树。

hdu 4081，次小生成树。

hdu 4126/4756，次小生成树。用 kruskal 会超时，需要结合 prim 和树形 DP。

hdu 4750，最小瓶颈生成树。

10.11　最　大　流

最大流问题(Maximum Flow Problem)是网络流中的基本问题，它是基于有向图的。最大流问题的解决有助于解决其他网络流问题，例如最小割、二分图匹配等。

最大流问题在生活中常见的原型是水流问题。hdu 1532 题描述了这个模型。

> **hdu 1532 “Drainage Ditches”**
>
> 约翰在农场建造了一套排水沟，以便下雨时把池塘的水排放到附近的溪流中。约翰还在每个水沟的入口安装了调节器，可以控制水流入该水沟的速度。
>
> 约翰不仅知道每个水沟每分钟可以运输多少加仑的水，而且还知道水沟的确切布局，水在这些水沟里相互进入和流动。对于任何给定的水沟，水只沿一个方向流动。水可能在某些水沟里兜圈子。
>
> 求源点 1(就是水塘)到终点 M(就是溪流)的最大流速。

在计算机网络中有带宽的概念，即每秒可传送的数据流量，和水流这个模型是一样的。

另外一个最大流模型的例子是道路的宽度。道路有单车道、双车道、四车道，同时能开行的车辆数量不同。这些不同道路的运输能力是不同的。注意这里需要假设所有车的速度都一样。

在 10.9 节中曾提到**“可加性参数”**和**“最小性参数”**。最大流问题的水流、带宽和宽度都是“最小性参数”。例如，一条路径上的最大水流由这条路径上水流容量最小的那条边决定，也就是说，由这条路径上的“瓶颈”决定。

最大流问题就是求两点间(分别称为源点、汇点)的最大流速，图中的任何点都可以作为它们的中转。在求最大流时需要满足以下 3 个性质：

(1) 流量守恒。从源点 s 流出的流量和到达汇点 t 的流量相等；其他所有中转点，流入

和流出相等。

(2) 反对称性。设从 u 到 v 的流量是 $f(u,v)$，v 到 u 的流量是 $f(v,u)$，那么 $f(u,v)=-f(v,u)$。

(3) 容量限制。每个边的实际流速不大于最大流速(把最大流速称为容量)。

算法需要搜索所有的点和边。

最大流算法有很多种，基本上分为两类：

(1) “增广路”算法。例如 Edmonds-Karp 算法、Dinic 算法。

(2) “预流推进”算法。例如 ISAP 算法。

Edmonds-Karp 算法比较容易，但是效率不高，在竞赛中一般使用 Dinic 算法和 ISAP 算法。

在学习有效的最大流算法之前，读者可以自己思考暴力法或者简单的贪心法。

10.11.1 Ford-Fulkerson 方法

Edmonds-Karp 算法是 Ford-Fulkerson 方法的一种实现。所谓 Ford-Fulkerson 方法，是一种非常容易理解的算法思想：

(1) 在初始的时候，所有边上的流量为 0。

(2) 找到一条从 s 到 t 的路径，按 3 个性质得到这条路径上的最大流，更新每个边的残留容量。残留容量在后续步骤中继续使用。

(3) 重复步骤(2)，直到找不到路径。

以图 10.27 为例[①]，图(a)中的斜体数字标出了每个边的容量，开始时每个边的流量是 0；图 10.27(b)是第 1 次迭代，找到了一条路径 $s \to a \to t$，画线数字是每个边的流量，斜体数字是残留容量；图 10.27(c)是第 2 次迭代，找到了一条路径 $s \to b \to t$，更新每个边的流量和残留容量。第 2 次迭代后没有新的路径，结束(注意：这里为了介绍思想，简化了过程；这实际上是**有错误**的，解释见下面的“残留网络”)。

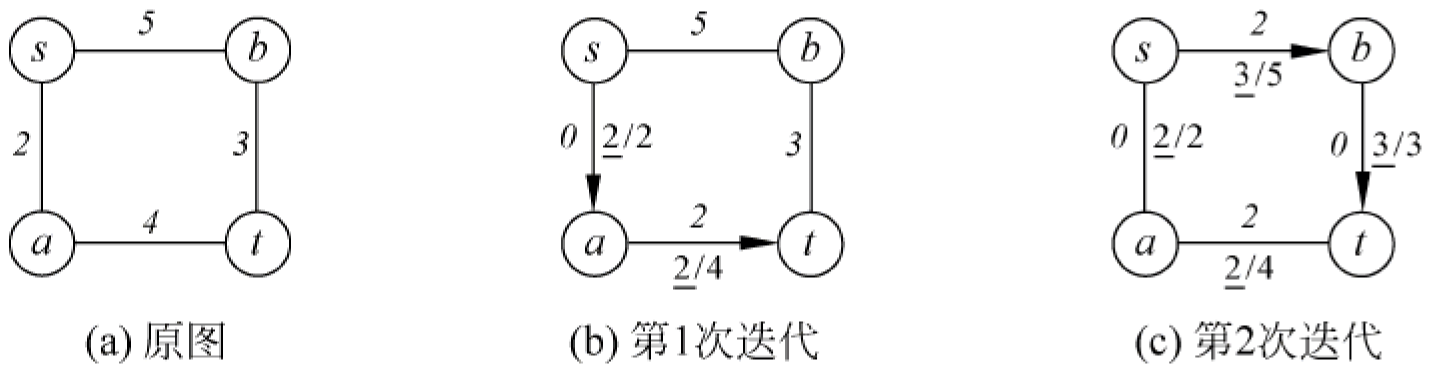


图 10.27 Ford-Fulkerson 方法示意

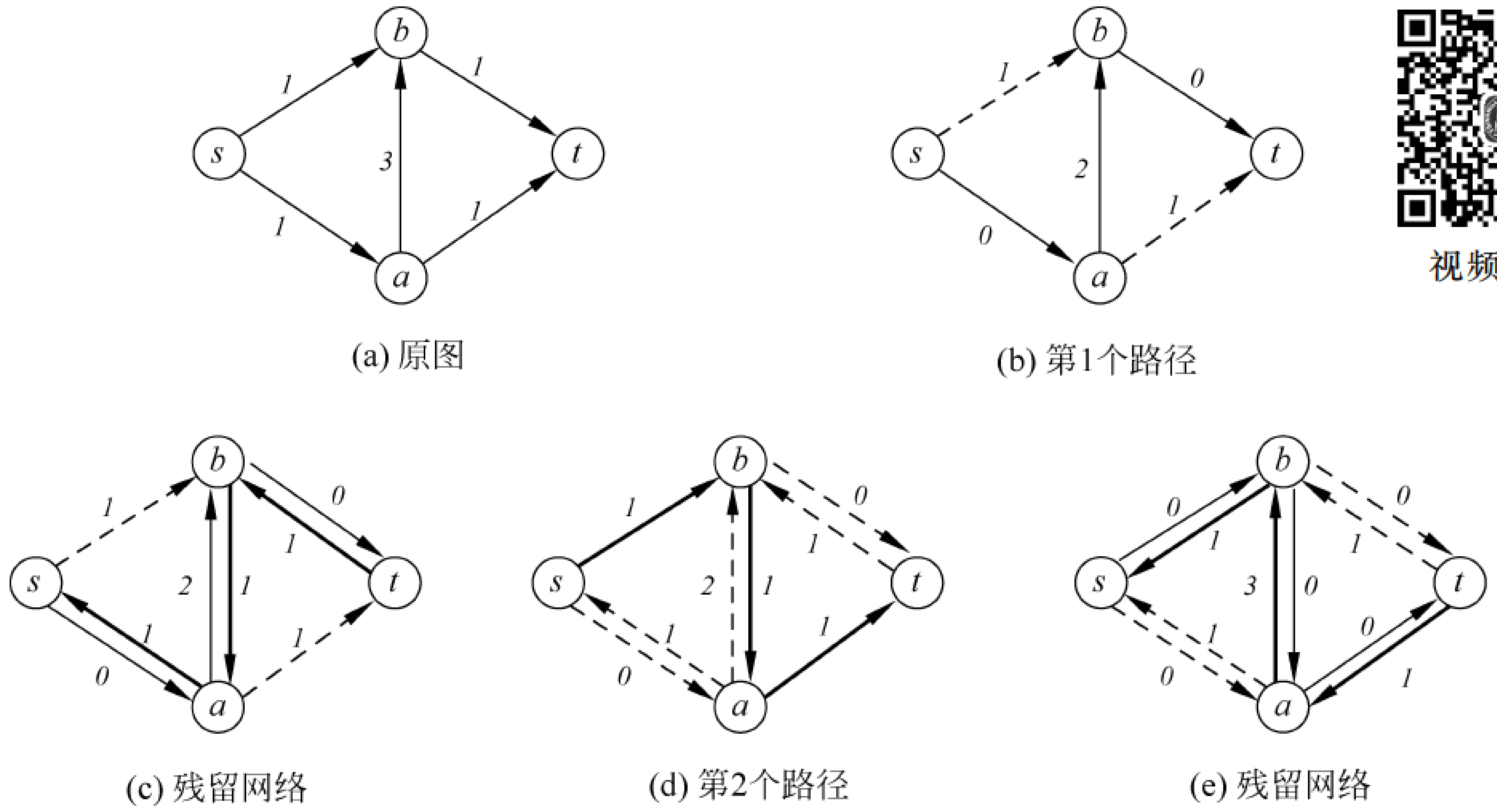
Ford-Fulkerson 方法基本上就是上述的思路。它有 3 个思想，也是后文将提到的“最大流最小割定理”的基础：

(1) 残留网络(residual network)。迭代后残留容量所产生的图，每次新的迭代在上一次的残留网络上进行。

但是，它实际上**并不是**图 10.27(a)、(b)、(c)中的斜体数字所表示的图，因为这个图在

① 这个例子过于简单，更完整的例子请参考《算法导论》，Thomas H. Cormen 等著，潘金贵等译，机械工业出版社，26.2 节，图 26-5。

迭代过程中损失了一些信息。请读者仔细分析图 10.28 所示的图解。


(a) 原图　(b) 第1个路径　(c) 残留网络　(d) 第2个路径　(e) 残留网络

视频讲解

图 10.28　残留网络

对于图 10.28(a)上的最大流,容易发现,在 *s-a-t*、*s-b-t* 这两条路径上最大流等于 2。

下面找一条路径。图 10.28(b)是搜到的第 1 个路径(读者可以想象 *s-b*、*a-t* 原来不存在水沟),产生的流量是 1;图上的数字是残留容量。如果在这个图上继续搜索路径,已经没有新路径。这显然是不对的。其原因是第 1 次搜索的结果影响了后续的路径搜索。那么如何消除这个影响?

图 10.28(c)是解决方法,在上一次的路径上补充反向路径,其值就是用过的流量 1,形成的新网络图就是残留网络。

残留网络的原理可以这样理解:在搜索新的增广路时,可能会经过以前的增广路使用过的水沟,而这个新路的水流可能与原来的水流相反,所以需要补上反向路径,让新的搜索有反向水流的机会。

图 10.28(d)是在图 10.28(c)的基础上搜到的第 2 个路径,这次结果是对的。

图 10.28(e)是最后的残留网络。此时,从 *s* 到 *t*,在残留网络上不存在新的路径,结束。为加深理解,请读者验证并思考:最后的残留网络,两点之间反向路径的值就是两点之间的实际流量。所以,可以利用残留网络输出最大流时各水沟中的实际流量。

残留网络和残留网络的反向路径是 Ford-Fulkerson 方法**最关键的技术**。

(2) 增广路(augmenting path)。在残留网络上找到的一条从 *s* 到 *t* 的路径。

(3) 割(cut)。Ford-Fulkerson 方法的正确性是最大流最小割定理的推论:一个流是最大流,当且仅当它的残留网络不包含增广路径时。

Ford-Fulkerson 方法的运行时间依赖于增广路径的搜索次数。虽然用 BFS 或者 DFS 都行,但是 DFS 这种深度搜索模式可能陷入长时间的迭代,图 10.29 是一个例子。

在图 10.29(b)和(c)中,很不幸地,DFS 选择了 *s-b-a-t* 和 *s-a-b-t* 这种绕路,接下来又反复选择这两个路径。在到达终点图 10.29(d)前,共迭代了约 200 次。

如果用 BFS,几次就够了。

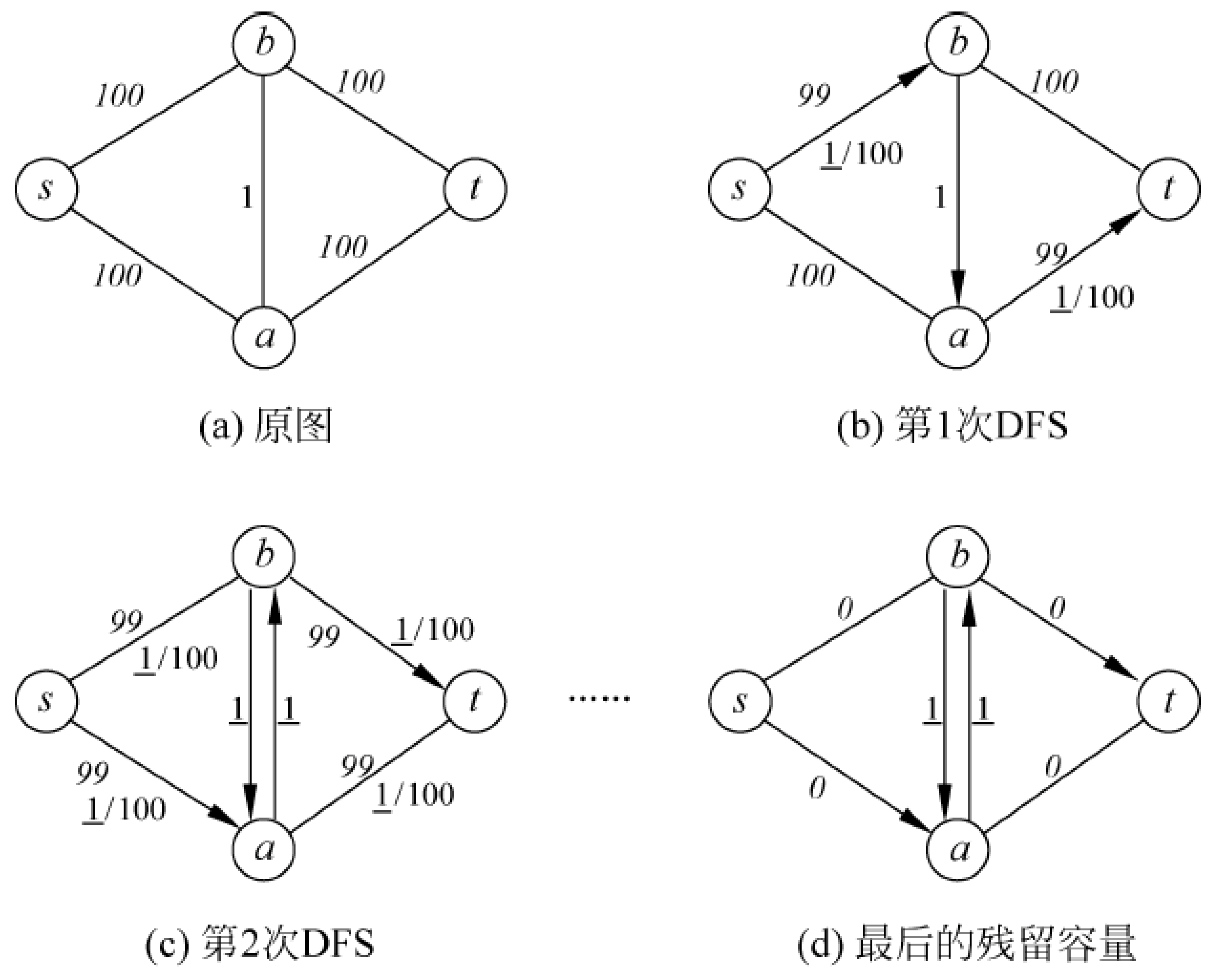


图 10.29　DFS 模式陷入长时间的迭代

10.11.2　Edmonds-Karp 算法

如果用 BFS 来计算增广路径，就是 Edmonds-Karp 算法。

复杂度：经过 $O(VE)$次 BFS 迭代，所有增广路被找到；一次 BFS 的时间是 $O(E)$，所以总时间是 $O(VE^2)$。

由于 Edmonds-Karp 算法的复杂度高，只能用于小图，所以用邻接矩阵存图就行了。

下面是 hdu 1532 题的代码，用矩阵 graph[][]存图，它同时也用于记录更新后的残留网络。

hdu 1532 题的代码

```
#include<bits/stdc++.h>
const int INF = 1e9;
const int maxn = 300;
using namespace std;
int n, m, graph[maxn][maxn], pre[maxn];
            //graph[][]不仅记录图，还是残留网络
int bfs(int s,int t){
    int flow[maxn];
    memset(pre, -1,sizeof pre);
    flow[s] = INF; pre[s] = 0;                    //初始化起点
    queue<int> Q;  Q.push(s);                     //起点入栈，开始 BFS
    while(!Q.empty()){
        int u = Q.front();  Q.pop();
        if(u==t) break;                           //搜到一个路径，这次 BFS 结束
        for(int i=1; i<=m; i++){                  //BFS 所有的点
            if(i!= s && graph[u][i]> 0 && pre[i] == -1){
                pre[i] = u;                       //记录路径
                Q.push(i);
```

```
                flow[i] = min(flow[u],graph[u][i]); //更新结点流量
            }
        }
    }
    if(pre[t] == -1) return -1;                     //没有找到新的增广路
    return flow[t];                                 //返回这个增广路的流量
}
int maxflow(int s, int t){
    int Maxflow = 0;
    while(1){
        int flow = bfs(s,t);
              //执行一次 BFS,找到一条路径,返回路径的流量
        if(flow == -1) break;                       //没有找到新的增广路,结束
        int cur = t;                                //更新路径上的残留网络
        while(cur!= s){                             //一直沿路径回溯到起点
            int father = pre[cur];                  //pre[]记录路径上的前一个点
            graph[father][cur] -= flow;             //更新残留网络：正向减
            graph[cur][father] += flow;             //更新残留网络：反向加
            cur = father;
        }
        Maxflow += flow;
    }
    return Maxflow;
}
int main(){
    while(~scanf("%d%d",&n,&m)){
        memset(graph,0,sizeof graph);
        for(int i = 0; i < n; i++){
            int u,v,w;
            scanf("%d%d%d",&u,&v,&w);
            graph[u][v] += w;                       //可能有重边
        }
        printf("%d\n",maxflow(1,m));
    }
    return 0;
}
```

最大流的建模问题。前面最大流的模型是基于有向图的,而且只有一个源点和一个汇点,但是题目所给的条件不一定这么严格,此时需要转换为下面的模型。

(1) 无向图转换为有向图。如果给的是无向图,可以把 u、v 之间的无向边变为(u,v)、(v,u)两个有向边,容量一样。u、v 的实际流量为两者的实际流量之差,即互相**抵消**,例如从 u 到 v 的流量是 10,从 v 到 u 的流量是 4,那么从 u 到 v 的流量是 10－4＝6。

(2) 多个源点和多个汇点。此时可以添加一个"超级源点"s 和一个"超级汇点"t。从 s 到每个源点都连一条有向边;从每个汇点都连一条边到 t。边的容量根据题目要求灵活指定。

在 10.13 节中,例题 poj 2135 "Farm Tour"就用到了这两个转换方法。在 10.14 节中有多源点、多汇点的情况。

10.11.3 Dinic 算法和 ISAP 算法

Edmonds-Karp 算法的效率低，在竞赛时若遇到规模较大的最大流问题，需要用高效的 Dinic 算法和 ISAP 算法。

Dinic 算法是对 Edmonds-Karp 算法的优化，时间复杂度理论上是 $O(V^2E)$，实际上更好，比 Edmonds-Karp 算法的 $O(VE^2)$ 强很多。

ISAP 算法的复杂度也是 $O(V^2E)$，但是比 Dinic 算法更好一些，更受欢迎。

Dinic 算法和 ISAP 算法[①]相当复杂，代码也比 Edmonds-Karp 算法长得多，在竞赛的时候靠自己写出来很困难。建议读者阅读有关资料，搞懂原理，学习其思想；然后找到合适的模板，特别是 ISAP 算法，学会使用它，在比赛的时候带上。

下面用最大流求解混合图的欧拉回路。

最大流算法是网络流算法的基础，它有很多应用。例如，最大流算法可用于判断和求解混合图的欧拉回路。请读者先回顾 10.5 节。

hdu 1956 "Sightseeing Tour"
给定一个图，其中同时存在有向边和无向边，问该图是否存在欧拉回路。

有向图存在欧拉回路的充要条件是所有点的度数为 0。把每个点连接的无向边改成有向边，看度数是否为 0。但是无向边很多，情况复杂，不能直接用暴力的方法做。

读者可以先思考，尝试用最大流方法解决。然后阅读下面的解题思路。

把所有的无向边任意定个方向，把这个包括原来的有向边和设定了方向的无向边的图称为初始图 G，然后计算每个点的度数。点 i 的度数 degree[i]＝出度－入度，有以下两种情况：

(1) 存在一个 degree[i]为奇数。如果把 i 的一个无向边改个方向，那么 degree[i]变为 degree[i]＋2 或 degree[i]－2，仍然是奇数，不会等于 0，所以不存在欧拉回路。

(2) 所有的 degree[i]全是偶数。可以把某个 i 的一个无向边改个方向，degree[i]变为 0。那么是否**所有的点的度数都能变为 0** 呢？可以借助最大流来判断。

下面用初始图 G 建一个新图 G'，在 G 中计算得到的 degree[i]也用于建图。首先把初始图 G 中原来的有向边删除，保留定向了的无向边。然后建一个源点 s，连接**所有的** degree[i]＞0 的点，边的容量为 degree[i]/2。建一个汇点 t，把所有 degree[i]＜0 的点连接到 t，容量为 degree[i]/2。其他 degree[i]＝0 的点就不用连接 s 和 t 了。所有没有连接 s 和 t 的边，容量都为 1。

求新网络 G' 的最大流。如果从 s 出发的所有边都满流，则存在欧拉回路。把所有的有流的边全部反向，把原图中的有向边再重新加入，就得到了一个有向欧拉回路。

上述算法正确吗？或者说，上述算法的结果能使得所有点的度数为 0 吗？

分 3 种情况观察：

---

① Dinic 算法和 ISAP 算法的对比：https://www.cnblogs.com/zhsl/archive/2012/12/03/2800092.html（永久网址：https://perma.cc/LAP9-QH83）。

(1) 观察源点 s 所连接的点 v,是否能得到 degree(v)=0 的结果。在图 10.30 所示的例子中,图 10.30(a)是初始图 G 的局部,v 在 G 中有 4 个边,degree[v]=4,其中虚线是有向边,在 G'中被删除了,剩下的 3 个实线边是原来的无向边,把方向定为出度。在图 10.30(b)中,加上源点 s,边(s,v)的容量是 degree[v]/2=2,v 的其他边的容量是 1。经过最大流的计算,如果(s,v)是满流 2,生成了图 10.30(c)中粗线条表示的流。把有流的边**反向**,得到图 10.30(d),可以发现,degree(v)=0,符合欧拉回路的要求。从这个图也能理解为什么把边(s,v)的容量设定为 degree[v]/2。

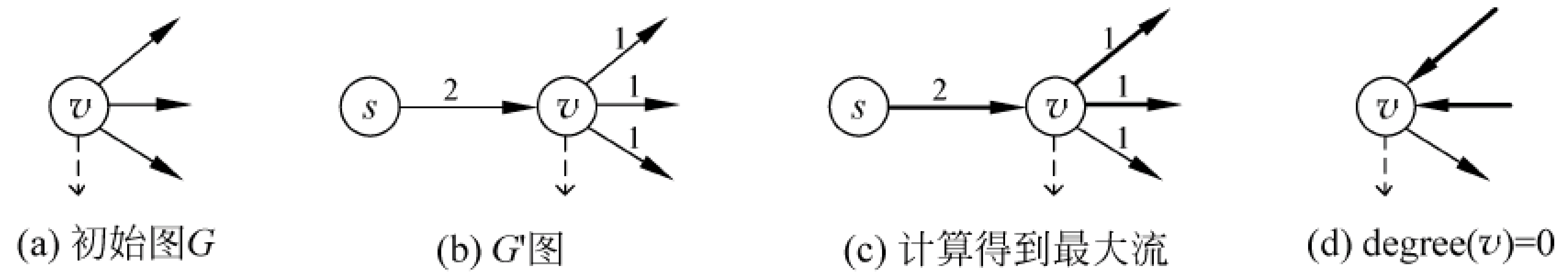


图 10.30 源点 s 连接的点 v

(2) 与汇点 t 连接的点,分析同上。

(3) 不与 s 和 t 连接的点 i,是否最后也有 degree(i)=0 的结果?这些点在初始图 G 中有 degree(i)=0。在 G'中计算最大流的路径时,如果增广路经过了点 i,那么肯定有一个进边的流和一个出边的流,把这两个边同时反向,仍然是一个进边和一个出边,仍保持 degree(i)=0。

hdu 1956 的数据比较大,需要用 Dinic 或 ISAP 算法。

【习题】

hdu 3549 “Flow Problem”,最大流入门题。

hdu 4280 “Island Transport”,数据规模为 $2 \leqslant N, M \leqslant 100\,000$。ISAP 模板题,用 Dinic 有可能超时。

hdu 3472 “HS BDC”。有 n 个单词,有的可以前后颠倒,看是否可以将 n 个单词首尾相连。混合图欧拉回路。

10.12 最 小 割

s-t 最小割是最大流的一个直接应用。

割(cut)和 s-t 割的概念:在有向图流网络 $G=(V,E)$中,割把图分成 S 和 $T=V-S$ 两部分,源点 $s \in S$,汇点 $t \in T$,这称为 s-t 割。

在图 10.31 中,边上的数字标出了流量和容量,s 和 t 之间的流量是 14。图中的虚线是一个割,把图分成了 S、T 两部分。

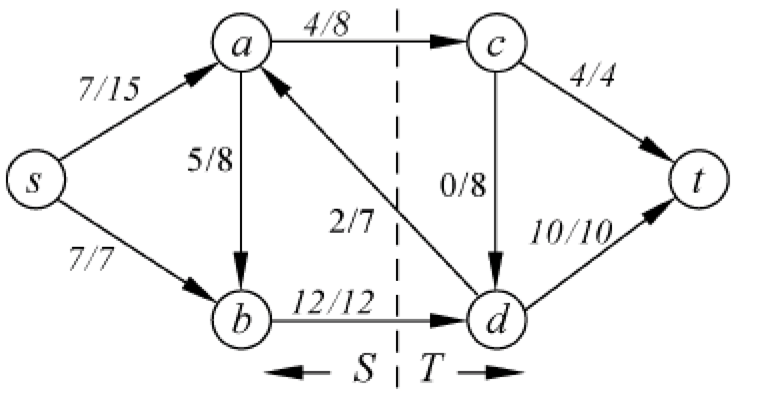


图 10.31 s-t 割

从 S 到 T,穿过割的净流量是 4+12−2=14。显然,在 s、t 之间做任意割,流经这个割的净流量都相等。

S 经过这个割到 T 的容量是 8+12=20,分别是边 ac 和 bd。也就是说,如果把边 ac 和 bd 去掉,S 中的水就

不能流到 T。注意在计算 S 到 T 的容量时不要算从 T 到 S 的反向容量。图中的虚线并不是一个最小割，读者可以观察最小割在哪里。

s-t 最小割问题是针对容量的，就是找到源点 s 和汇点 t 之间容量最小的割。

最小割问题可以形象地理解为：为了不让水从 s 流向 t，怎么破坏水沟代价最小？被破坏的水沟必然是从 s 到 t 的单向水沟。

最大流最小割定理：源点 s 和汇点 t 之间的最小割等于 s 和 t 之间的最大流[①]。

需要注意的是，定理中的最大流是指流量，而最小割是指容量。

全局最小割：把 s-t 最小割问题扩展到全局，有全局最小割问题。

简单的思路：可以利用最小割最大流定理，即枚举每个点当作汇点，计算出它的最大流，然后在所有点的最大流中取最小值。

但是这样做的复杂度很高，枚举汇点要 $O(V)$，Dinic 或 ISAP 算法的复杂度是 $O(V^2E)$，总复杂度是 $O(V^3E)$。

解决此类问题需要用 Stoer-Wagner 算法，由于题目比较罕见，本书不展开介绍。读者可以通过 poj 2914 "Minimum Cut"来了解。

【习题】

普通最小割问题的编码就是最大流算法。在对问题正确建模之后，用最大流的算法思路解决。

hdu 3251 "Being a Hero"，最小割。

poj 1815 "Friendship"，最小割。

10.13 最小费用最大流

在最大流网络中，每条边只有一个限制条件，例如容量、带宽等，这是"最小性参数"，现在加上一个新的限制条件，例如费用，这是"可加性参数"。在两个限制条件的基础上引出了最小费用最大流问题：当流量为 F 时，求费用最小的流；如果没有指定 F，就是求最大流时的最小费用。

有两种思路：

(1) 先求一个最大流，然后不断优化得到最小费用流。首先用最大流算法得到一个最大流，然后检查边的情况，看是否有费用更小同时也能满足最大流的边，如果有，就进行调整，得到一个新的最大流。经过多次迭代，直到所有边都无法调整，就得到了最小费用最大流。

(2) 从零流开始，每次增加一个最小费用路径，经过多次增广，直到无法再增加路径，就得到了最大流。

思路(2)更容易理解和操作，它是网络流问题和最短路径问题的结合，其算法也是最大流算法和最短路径算法的结合。

---

① 证明见《算法导论》，Thomas H. Cormen 等著，潘金贵等译，机械工业出版社，定理 26.7。

最短路径算法有 Bellman-Ford 算法、Dijkstra 算法等，是否都能用？如果边的费用权值有负数，只能选择 Bellman-Ford 算法（或 SPFA 算法）。在最小费用最大流算法中，由于残留网络用到了反向边，所以肯定会出现负权边[①]，在本节的例题中会说明这一问题。

最小费用最大流的解决方法是 Ford-Fulkerson 方法+Bellman-Ford 算法（SPFA 算法）。

回顾最大流的 Ford-Fulkerson 方法，它的主要操作是在残留网络上不断寻找增广路径。如果用 BFS 求增广路，就是 Edmonds-Karp 算法。BFS 求增广路是很盲目的，它不会区分增广路的“好坏”。

如何找一条“好”的增广路？如果不用 BFS，而是改用 Bellman-Ford 算法（SPFA 算法），每次在残留网络上找增广路时都找费用最小的路径，就会得到一条“好”的、费用最低的路径。不断用 Bellman-Ford 算法（SPFA 算法）求增广路，直到满足题目要求的流量 F，最后得到一个流量为 F 并且费用最小的流。

上述的算法思想是否正确？可以简单思考如下：如果经过上述步骤得到的不是最小费用流，说明在残留网络上还存在费用更小的路径，这与前面步骤中已经计算了最小路径相矛盾[②]。

算法的复杂度是多少？找一次增广路，这个路径上至少有一个流量；总流量为 F，最多需要找 F 次增广路；每次使用 Bellman-Ford 算法找增广路，一次 Bellman-Ford 的时间是 $O(VE)$，所以总时间是 $O(FVE)$。

对于下面的例题，请读者先思考，再看答案。

poj 2135 “Farm Tour”

一个无向图，有 N 个地点、M 条边。一个人从 1 号点走到 N 号点，再从 N 号点走回 1 号点，每条路只能走一次。求来回的总长度最短的路线。

输入：第 1 行是两个整数 N、M；后面有 M 行，每行有 3 个整数，描述一个边的两个端点和边的长度。$1 \leqslant N \leqslant 1000$，$1 \leqslant M \leqslant 10\,000$。

输出：来回总长度最短的路径长度。

题目的测试数据确保存在来回的不重复路径。

根据题意分析，这是个无向图，从 1 走到 N 和从 N 走到 1 是一样的，那么题目转换为从 1 号点到 N 号点至少有两条不同路线，找其中两条，使它们的总长度最短。

刚看到这一题的时候，读者可能觉得很简单：先求第一个最短路径，然后把走过的路删除，再算一次最短路径。

然而这样做是错误的。例如图 10.32，找 a 到 d 的两条路。图中确实存在两条路，但是直接算两次最短路径却找不到这两条路：第一条最短路径是 a-c-b-d，有 3 个边，如果删除这 3 个边，图就断开了，无法继续找第二条路。

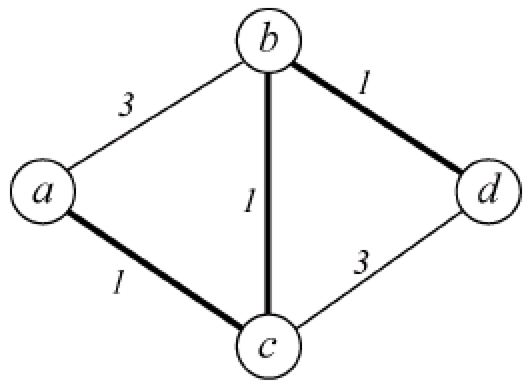


图 10.32 寻找 a 和 d 之间的两条路径

---

① 通过导入“势”的概念，可以在最小费用最大流算法中用 Dijkstra 算法，从而降低算法复杂度。请参考《挑战程序设计竞赛》（秋叶拓哉），225 页，“3.5.6 最小费用流”。

② 算法正确性的具体证明参考《挑战程序设计竞赛》（秋叶拓哉），225 页，“3.5.6 最小费用流”。

这个例子是从前面最大流的"残留网络"的例子引用过来的。这个例子说明本题和最大流有关系。

这一题实际上是一道最小费用最大流的裸题。建模如下：

把每条边的流量设为1，表示每条边只能用1次，把边的长度看成每个边的费用。在图中添加一个"超级源点"s和一个"超级汇点"t，s到1有一个长度为0、容量为2的边；N到t有一个长度为0、容量为2的边。在经过这个建模之后，原题中求两条最短路径的费用等价于求源点s和汇点t的最小费用最大流[①]。

分析复杂度，最小费用最大流的复杂度是$O(FVE)$，$F=2$，$V=1000$，$E=10\ 000$，$FVE=$2000万，正好满足。

下面的最小费用最大流程序综合了SPFA算法和最大流算法，基本上套用了前面讲解过的模板。其中需要特别注意的是图的初始化，即如何把无向图转为有向图。

无向图的两个点(u,v)之间只有1个边，本题把它变成了4个边。

首先把无向边(u,v)分成有向边(u,v)和(v,u)。

然后把它们各分成两个边。例如有向边(u,v)变成了一个正向的费用为cost、容量为capacity的边，以及一个反向的费用为−cost、容量为0的边。这样做和最大流中的残留网络是同样的道理，相当于一次增广之后生成的残留网络。如果读者不能理解，请回顾最大流中"反向路径"的相关内容。

从这个例子可以看出，边的权值会出现负数，所以不能用Dijkstra算最短路径，只能用SPFA。

poj 2135 程序（邻接表存图+SPFA+最大流）

```
#include <stdio.h>
#include <algorithm>
#include <cstring>
#include <queue>
using namespace std;
const int INF = 0x3f3f3f3f;
const int N = 1010;
int dis[N], pre[N], preve[N];
//dis[i]记录起点到i的最短距离.pre和preve见下面的注释
int n, m;
struct edge{
    int to, cost, capacity, rev;                    //rev用于记录前驱点
    edge(int to_,int cost_,int c,int rev_){
          to = to_; cost = cost_; capacity = c; rev = rev_;}
};
vector<edge> e[N];                                  //e[i]: 存第i个结点连接的所有的边
void addedge(int from,int to,int cost,int capacity){  //把1个有向边再分为两个
    e[from].push_back(edge(to, cost, capacity, e[to].size()));
    e[to].push_back(edge(from, -cost, 0, e[from].size()-1));
}
bool spfa(int s, int t, int cnt){                  //套SPFA模板
```

① 从这一题的建模过程可以看出，单源最短路径问题是费用流问题的一个特殊情况。把每个边的容量设为1，添加一个源点s，s到起点的边容量是1、费用是0，那么s到终点的最小费用最大流就是最短路径。

```
    bool inq[N];
    memset(pre, -1, sizeof(pre));
    for(int i = 1; i <= cnt; ++i) { dis[i] = INF; inq[i] = false; }
    dis[s] = 0;
    queue<int> Q;
    Q.push(s);
    inq[s] = true;
    while(!Q.empty()){
        int u = Q.front();
        Q.pop();
        inq[u] = false;
        for(int i = 0; i < e[u].size(); i++)
            if(e[u][i].capacity > 0){
                int v = e[u][i].to, cost = e[u][i].cost;
                if(dis[u] + cost < dis[v]){
                    dis[v] = dis[u] + cost;
                    pre[v] = u;                 //v 的前驱点是 u
                    preve[v] = i;               //u 的第 i 个边连接 v 点
                    if(!inq[v]){
                        inq[v] = true;
                        Q.push(v);
                    }
                }
            }
    }
    return dis[t] != INF;                       //s 到 t 的最短距离(或者最小费用)是 dis[t]
}
int mincost(int s, int t, int cnt){             //基本上是套最大流模板
    int cost = 0;
    while(spfa(s, t, cnt)){
        int v = t, flow = INF;                  //每次增加的流量
        while(pre[v] != -1){                    //回溯整个路径,计算路径的流
            int u = pre[v], i = preve[v];
                    //u 是 v 的前驱点,u 的第 i 个边连接 v
            flow = min(flow, e[u][i].capacity);
                    //所有边的最小容量就是这条路的流
            v = u;                              //回溯,直到源点
        }
        v = t;
        while(pre[v] != -1){                    //更新残留网络
            int u = pre[v], i = preve[v];
            e[u][i].capacity -= flow;           //正向减
            e[v][e[u][i].rev].capacity += flow; //反向加,注意 rev 的作用
            v = u;                              //回退,直到源点
        }
        cost += dis[t] * flow;
                    //费用累加.如果程序需要输出最大流,在这里累加 flow
    }
    return cost;                                //返回总费用
}
int main(){
    while(~scanf("%d%d", &n, &m)){
        for(int i = 0; i < N; i++)  e[i].clear(); //清空待用
        for(int i = 1; i <= m; ++i){
```

```
            int u, v, w;
            scanf("%d%d%d", &u, &v, &w);
            addedge(u, v, w, 1);            //把 1 个无向边分为 2 个有向边
            addedge(v, u, w, 1);
        }
        int s = n+1, t = n+2;
        addedge(s, 1, 0, 2);                //添加源点
        addedge(n, t, 0, 2);                //添加汇点
        printf("%d\n", mincost(s, t, n+2));
    }
    return 0;
}
```

【习题】

hdu 3376 “Matrix Again”,费用流裸题。

hdu 3667 “Transportation”。

hdu 5520 “Number Link”。

10.14 二分图匹配

视频讲解

二分图:把无向图 $G=(V, E)$分为两个集合 V_1、V_2,所有的边都在 V_1 和 V_2 之间,而 V_1 或 V_2 的内部没有边。V_1 中的一个点与 V_2 中的一个点关联,称为一个匹配。

一个图是否为二分图,一般用“染色法”进行判断。用两种颜色对所有顶点进行染色,要求一条边所连接的两个相邻顶点的颜色不相同。染色结束后,如果所有相邻顶点的颜色都不相同,它就是二分图。

一个图是二分图,当且仅当它不含边的数量为奇数的圈。读者可以画图理解这一点。

常见的二分图匹配问题有两种。

(1) 无权图,求包含边数最多的匹配,即二分图的最大匹配。本节讲解这个问题。

(2) 带权图,求边权之和尽量大的匹配。使用 KM 算法,本书没有涉及。

1. 二分图最大匹配问题

可以将二分图最大匹配问题转化为求最大流问题的思想来解决。不过在竞赛时一般不用标准的最大流模板,而是使用更简单的匈牙利算法。

二分图最大匹配的原型见下面的题目。

hdu 2063 “过山车”

大家去坐过山车。过山车的每一排只有两个座位,并且必须一男一女配对坐。但是,每个女孩有各自的想法,比如 Rabbit 只愿意和 XHD 或 PQK 坐,Grass 只愿意和 linle 或 LL 坐,等等。boss 刘决定,只让能配对的人坐过山车。当然,能配对的人越多越好。问最多有多少对组合可以坐上过山车?

2. 用最大流求解二分图匹配

二分图最大匹配问题可以转化为最大流问题：把每个边都改为有向边，流量都是1；在V_1上加一个人为的源点s，它连接V_1的所有点；在V_2上加一个人为的汇点t，它连接V_2的所有点，那么s、t之间的最大流就是最大二分图匹配。

原理很直观。在图10.33中，$V_1=\{a,b,c\}$是女生，$V_2=\{x,y,z\}$是男生。例如a点，流入a的流量是1，那么从a流出的只能是1，也就是说a只能匹配$\{x,y,z\}$中的一个。从V_1到V_2的流量和从s到t的流量相等。

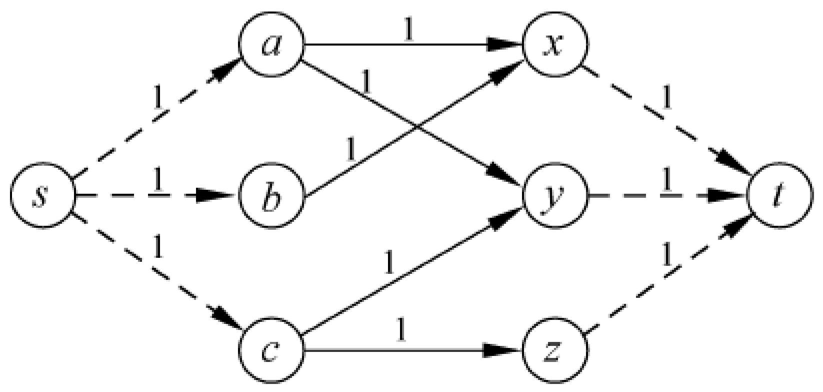


图10.33　二分图匹配和最大流

下面用最大流来求解。读者可以用图10.34复习最大流的Ford-Fulkerson方法，主要是对残留网络的操作。

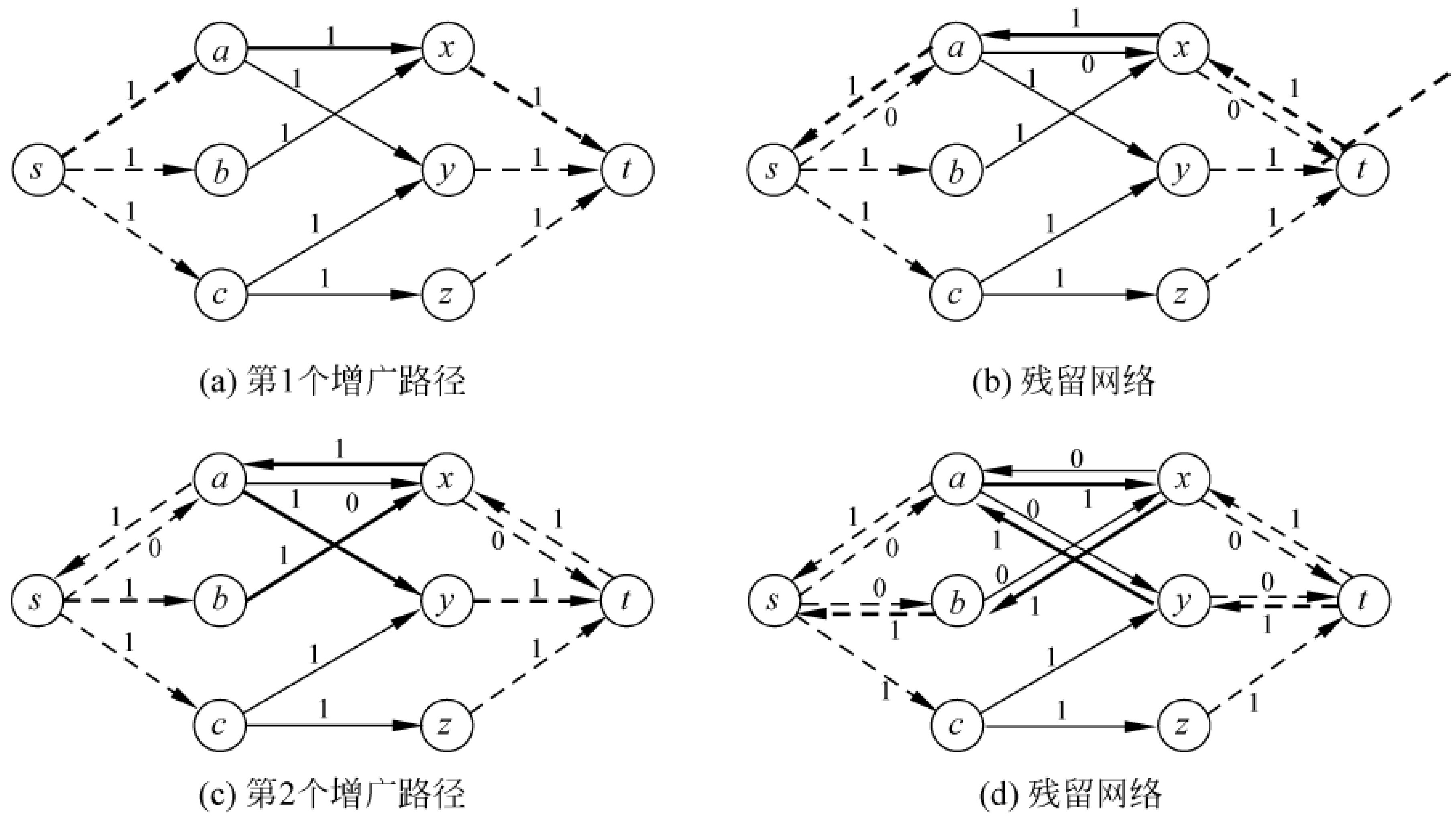


图10.34　用最大流来求解

(1) 找到第1个增广路径，找到匹配a-x，见图10.34(a)。

(2) 更新残留网络，见图10.34(b)。

(3) 找到第2个增广路径，找到匹配a-y、b-x。在这一步把原来的配对a-x改为a-y，以成全b-x的配对，这就是残留网络的作用，见图10.34(c)。

(4) 更新残留网络，见图10.34(d)。

后面的步骤请读者做练习。

3. 匈牙利算法

匈牙利算法可以看成最大流的一个特殊实现。

由于二分图是一个很简单的图，并不需要按上面的图解做标准的最大流，可以进行简化。

(1) 从上面的图解中发现对s和t的操作是多余的，直接从a、b、c开始找增广路径就可以了。

(2) 残留网络上的增广路需要覆盖完整的路径，如果在二分图中只进行$\{a,b,c\}$到$\{x,y,z\}$的局部操作，将简化很多。

下面是 hdu 2063 的程序。

hdu 2063 匈牙利算法(邻接矩阵)

```
#include <bits/stdc++.h>
using namespace std;
int G[510][510];
int match[510], reserve_boy[510];           //匹配结果在 match[ ]中
int k, m_girl, n_boy;
bool dfs(int x){                            //找一个增广路径,即给女孩 x 找一个配对男孩
    for(int i = 1; i <= n_boy; i++)
        if(!reserve_boy[i] && G[x][i]){
            reserve_boy[i]  = 1;            //预定男孩 i,准备分给女孩 x
            if(!match[i] || dfs(match[i])){
    //有两种情况: (1)如果男孩 i 还没配对,就分给女孩 x;
    //(2)如果男孩 i 已经配对,尝试用 dfs()更换原配女孩,以腾出位置给女孩 x
                match[i]  =  x;
                        //配对成功.如果原来有配对,更换成功.现在男孩 i 属于女孩 x
                return true;
            }
        }
    return false;                           //女孩 x 没有喜欢的男孩,或者更换不成功
}
int main(){
    while(scanf("%d",&k)!= EOF && k){
        scanf("%d%d",&m_girl,&n_boy);
        memset(G,0,sizeof(G));
        memset(match,0,sizeof(match));
        for(int i = 0;i < k;i++){
            int a,b;
            scanf("%d%d",&a,&b);
            G[a][b] = 1;
        }
        int sum = 0;
        for(int i = 1; i <= m_girl; i++){      //为每个女孩找配对
            memset(reserve_boy,0,sizeof(reserve_boy));
            if(dfs(i))   sum++;
                //第 i 个女孩配对成功,这个配对后面可能会更换,但是保证她能配对
        }
        printf("%d\n",sum);
    }
    return 0;
}
```

上述程序用邻接矩阵，找一次增广路径的时间复杂度为 $O(V^2)$，总时间为 $O(V^3)$；空间复杂度为 $O(V^2)$。

改用邻接表存图可以加快搜索速度。找一次增广路径的时间复杂度为 $O(V+E)$，总时

间为 $O(VE)$；空间复杂度为 $O(V+E)$。读者可以练习把上述程序改成用邻接表。

【习题】

hdu 1083 “Courses”,简单题。

hdu 3729 “I'm Telling the Truth”,简单题。

hdu 5727 “Necklace”。

hdu 3605 “Escape”,二分图多重匹配。

10.15 小　结

本章讲解了很多图论问题,关键的知识点总结如下。

(1) 图的存储：牢固掌握图的邻接矩阵、邻接表、链式前向星 3 种存储方法。

(2) BFS 和 DFS 在图问题中的关键作用：DFS 对图的遍历过程。

(3) 拓扑排序：BFS 和 DFS 的直接应用。

(4) 欧拉路：DFS 的直接应用。

(5) 无向图连通性：缩点的方法。

(6) 有向图连通性：DFS 的深度应用。

(7) 2-SAT 问题：强连通分量和拓扑排序的应用。

(8) 最短路径：透彻掌握各种最短路径算法的思想、数据结构、编程、应用环境。

(9) 最小生成树：贪心法思想的应用。

(10) 最大流：网络流的基础问题；残留网络、增广路的方法。请读者透彻掌握。

(11) 最小割：问题建模。

(12) 最小费用最大流：最短路径和最大流的结合。

(13) 二分图匹配：最大流思想的应用。

第 11 章　计 算 几 何

✍二维几何基础

✍圆

✍三维几何

✍几何模板

几何类题目是算法竞赛中的一个大类考点，涉及的知识点有平面几何、解析几何、计算几何等。

几何题的代码一般比较长，有时甚至有 200 多行，而且逻辑往往也比较复杂，是典型的考查参赛人员编码能力的题型。

如果要选出带到赛场的必备模板，其中一定会包括几何模板。很多有经验的老队员说："做几何题，模板很重要，要高度可靠！"因此，在平时的训练过程中认真总结模板，融会贯通，才能在赛场上灵活地使用它们。

11.1　二维几何基础

计算几何中的坐标值一般是实数，在编程时用 double 类型，不用精度较低的 float 类型。double 类型读入时用%lf 格式，输出时用%f 格式。

视频讲解

在进行浮点数运算时会产生精度误差，为了控制精度，可以设置一个偏差值 eps(epsilon)，eps 要大于浮点运算结果的不确定量，一般取 10^{-8}。如果 eps 取 10^{-10}，可能会有问题，例如 11.2.1 节中提到的 hdu 5572 题，用 10^{-10} 会返回 Wrong Answer。

判断一个浮点数是否等于 0，不能直接用"==0"来判断，而是用 sgn()函数判断是否小于 eps。在比较两个浮点数时，也不能直接用"=="判断是否相等，而是用 dcmp()函数判断是否相等。

```
const double pi = acos(-1.0);             //高精度圆周率
const double eps = 1e-8;                  //偏差值,有时用 1e-10
int sgn(double x){                        //判断 x 是否等于 0
    if(fabs(x) < eps)  return 0;
    else return x<0?-1:1;
}
int dcmp(double x, double y){             //比较两个浮点数: 0 为相等; -1 为小于; 1 为大于
    if(fabs(x - y) < eps) return 0;
    else return x<y ?-1:1;
}
```

11.1.1 点和向量

1. 点

二维平面中的点用坐标(x, y)来表示。

```
struct Point{
    double x,y;
    Point(){}
    Point(double x,double y):x(x),y(y){}
};
```

2. 两点之间的距离

(1) 把两点看成直角三角形的两个顶点,斜边就是两点的距离。用库函数 hypot()计算直角三角形的斜边长。

```
double Distance(Point A, Point B){return hypot(A.x-B.x,A.y-B.y);}
```

(2) 或者用 sqrt()函数计算。

```
double Dist(Point A,Point B){
    return sqrt((A.x-B.x)*(A.x-B.x) + (A.y-B.y)*(A.y-B.y));
}
```

3. 向量

有大小、有方向的量称为向量(矢量),只有大小没有方向的量称为标量。

用平面上的两个点可以确定一个向量,例如用起点 P_1 和终点 P_2 表示一个向量。不过,为了简化描述,可以把它平移到原点,把向量看成从原点(0,0)指向点(x,y)的一个有向线段,如图 11.1 所示。向量的表示在形式上与点的表示完全一样,可以用点的数据结构来表示向量:

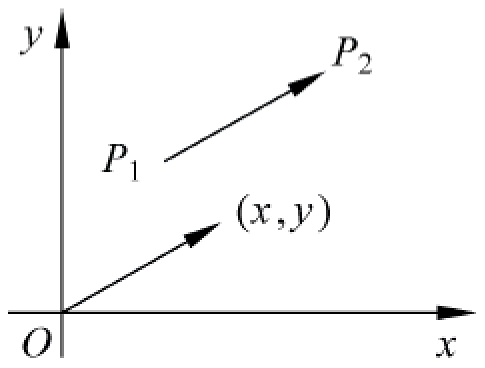


图 11.1 向量

```
typedef Point Vector;
```

注意,向量并不是一个有向线段,只是表示方向和大小,所以向量平移后仍然不变。

视频讲解

4. 向量的运算

在 struct Point 中,对向量运算重载运算符。

(1) 加:点与点的加法运算没有意义;点与向量相加得到另一个点;向量与向量相加得到另外一个向量。

```
Point operator + (Point B){return Point(x+B.x,y+B.y);}
```

(2) 减:两个点的差是一个向量;向量 **A** 减 **B** 得到由 **B** 指向 **A** 的向量。

```
Point operator - (Point B){return Point(x-B.x,y-B.y);}
```

向量的加法和减法示意图如图 11.2 所示。

图 11.2　向量的加法和减法

(3) 乘：向量与实数相乘得到等比例放大的向量。

```
Point operator * (double k){return Point(x*k,y*k);}
```

(4) 除：向量与实数相除得到等比例缩小的向量。

```
Point operator / (double k){return Point(x/k,y/k);}
```

(5) 等于：

```
bool operator == (Point B){return sgn(x - B.x) == 0 && sgn(y - B.y) == 0;}
```

11.1.2　点积和叉积

向量的基本运算是点积和叉积，计算几何的各种操作几乎都基于这两种运算。

1. 点积(Dot product)

记向量 **A** 和 **B** 的点积为 **A** · **B**，定义如下：

$$\boldsymbol{A}\cdot\boldsymbol{B}=|\boldsymbol{A}||\boldsymbol{B}|\cos\theta$$

其中 θ 为 **A**、**B** 之间的夹角。点积的几何意义为 **A** 在 **B** 上的投影长度乘以 **B** 的模长。点积的几何表示如图 11.3 所示。

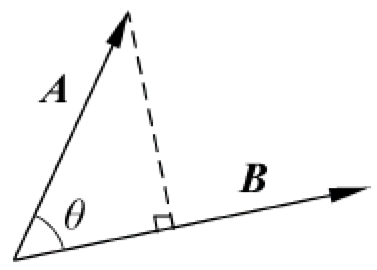


图 11.3　点积的几何表示

在编程时计算点积并不需要知道 θ。如果已知 **A**=(**A**.x，**A**.y)，**B**=(**B**.x，**B**.y)，那么有：

$$\boldsymbol{A}\cdot\boldsymbol{B}=\boldsymbol{A}.x*\boldsymbol{B}.x+\boldsymbol{A}.y*\boldsymbol{B}.y$$

下面推导这个公式。设 θ1 是 **A** 与 x 轴的夹角，θ2 是 **B** 与 x 轴的夹角，向量 **A** 与 **B** 的夹角 θ 等于 θ1−θ2，那么有：

$$\begin{aligned}&\boldsymbol{A}.x*\boldsymbol{B}.x+\boldsymbol{A}.y*\boldsymbol{B}.y\\&=(|\boldsymbol{A}|*\cos\theta1)*(|\boldsymbol{B}|*\cos\theta2)+(|\boldsymbol{A}|*\sin\theta1)*(|\boldsymbol{B}|*\sin\theta2)\\&=|\boldsymbol{A}||\boldsymbol{B}|(\cos\theta1*\cos\theta2+\sin\theta1*\sin\theta2)\\&=|\boldsymbol{A}||\boldsymbol{B}|(\cos(\theta1-\theta2))\\&=|\boldsymbol{A}||\boldsymbol{B}|\cos\theta\end{aligned}$$

求 **A**、**B** 点积的代码如下：

```
double Dot(Vector A,Vector B){return A.x*B.x+A.y*B.y;}
```

2. 点积的应用

1) 判断 **A** 与 **B** 的夹角是钝角还是锐角

点积有正负，利用正负号可以判断向量的夹角：

若 dot(**A**,**B**)>0，**A** 与 **B** 的夹角为锐角；

若 dot($\boldsymbol{A}$,$\boldsymbol{B}$)<0,$\boldsymbol{A}$ 与 $\boldsymbol{B}$ 的夹角为钝角;

若 dot($\boldsymbol{A}$,$\boldsymbol{B}$)=0,$\boldsymbol{A}$ 与 $\boldsymbol{B}$ 的夹角为直角。

2) 求向量 $\boldsymbol{A}$ 的长度

```
double Len(Vector A){return sqrt(Dot(A,A));}
```

或者是求长度的平方,避免开方运算:

```
double Len2(Vector A){return Dot(A,A);}
```

3) 求向量 $\boldsymbol{A}$ 与 $\boldsymbol{B}$ 的夹角大小

```
double Angle(Vector A,Vector B){return acos(Dot(A,B)/Len(A)/Len(B));}
```

3. 叉积(Cross product)

叉积是比点积更常用的几何概念。它的计算公式如下:

$$\boldsymbol{A}\times\boldsymbol{B}=|\boldsymbol{A}||\boldsymbol{B}|\sin\theta$$

θ 表示向量 $\boldsymbol{A}$ 旋转到向量 $\boldsymbol{B}$ 所经过的夹角。

两个向量的叉积是一个带正负号的数值。$\boldsymbol{A}\times\boldsymbol{B}$ 的几何意义为向量 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 形成的平行四边形的“有向”面积,这个面积是有正负的。叉积的正负符合“右手定则”,读者可以用图 11.4 中的正负情况帮助理解。

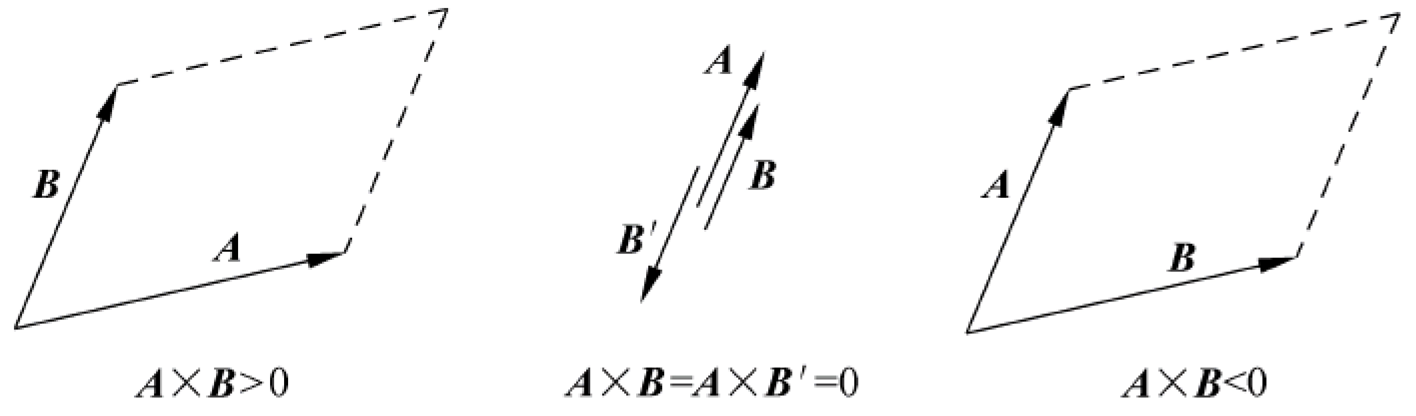


图 11.4　叉积与叉积的正负

用以下程序计算向量 $\boldsymbol{A}$、$\boldsymbol{B}$ 的叉积 $\boldsymbol{A}\times\boldsymbol{B}$:

```
double Cross(Vector A,Vector B){return A.x * B.y - A.y * B.x;}
```

对于其正确性,读者可以用前文证明点积的推导方法来证明。

注意函数 Cross()中的 $\boldsymbol{A}$、$\boldsymbol{B}$ 是有顺序的,叉积有正负,$\boldsymbol{A}\times\boldsymbol{B}$ 与 $\boldsymbol{B}\times\boldsymbol{A}$ 相反。

叉积有正负,这个性质使得叉积能用于很多有用的场合。

4. 叉积的基本应用

下面给出叉积的几个基本应用。对于其他应用,例如求两个线段的方向关系、求多边形的面积等,将在后文讲解。

1) 判断向量 $\boldsymbol{A}$、$\boldsymbol{B}$ 的方向关系

若 $\boldsymbol{A}\times\boldsymbol{B}>0$,$\boldsymbol{B}$ 在 $\boldsymbol{A}$ 的逆时针方向;

若 $\boldsymbol{A}\times\boldsymbol{B}<0$,$\boldsymbol{B}$ 在 $\boldsymbol{A}$ 的顺时针方向;

若 $\boldsymbol{A}\times\boldsymbol{B}=0$,$\boldsymbol{B}$ 与 $\boldsymbol{A}$ 共线,可能是同方向的,也可能是反方向的。

2）计算两向量构成的平行四边形的有向面积

3个点 **A**、**B**、**C**，以 **A** 为公共点，得到两个向量 **B**－**A** 和 **C**－**A**，它们构成的平行四边形的面积如下：

```
double Area2(Point A,Point B,Point C){return Cross(B-A, C-A);}
```

如果以 **B** 或 **C** 为公共点构成平行四边形，面积是相等的，但是正负不一样。

3）计算3点构成的三角形的面积

3个点 **A**、**B**、**C** 构成的三角形的面积等于平行四边形面积 Area2(**A**,**B**,**C**)的1/2。

4）向量旋转

使向量(x,y)绕起点逆时针旋转，设旋转角度为θ，那么旋转后的向量(x',y')如下：

$$x' = x\cos\theta - y\sin\theta$$

$$y' = x\sin\theta + y\cos\theta$$

代码如下，向量 **A** 逆时针旋转的角度为 rad：

```
Vector Rotate(Vector A, double rad){
    return Vector(A.x*cos(rad) - A.y*sin(rad), A.x*sin(rad) + A.y*cos(rad));
}
```

特殊情况是旋转90°。

逆时针旋转90°：Rotate(A, pi/2)，返回 Vector($-A.y$, $A.x$)；

顺时针旋转90°：Rotate(A, $-$pi/2)，返回 Vector($A.y$, $-A.x$)。

有时需要求单位法向量，即逆时针转90°，然后取单位值。代码如下：

```
Vector Normal(Vector A){return Vector(-A.y/Len(A), A.x/Len(A));}
```

5）用叉积检查两个向量是否平行或重合

```
bool Parallel(Vector A, Vector B){return sgn(Cross(A,B)) == 0;}
```

11.1.3 点和线

1. 直线的表示

直线有多种表示方法，用户在编程时可以灵活使用这些方法：

（1）用直线上的两个点来表示。

（2）$ax+by+c=0$，普通式。

（3）$y=kx+b$，斜截式。

（4）$P=P_0+vt$，点向式。也就是用 P_0 和 v 来表示直线 P，t 是变量，可以取任意值。$P_0(x_0,y_0)$是直线上的一个点；v 是方向向量，给定两个点 A、B，那么 $v=B-A$。

点向式非常便于计算机处理，也能方便地表示射线、线段：

如果 t 无限制，P 是直线；

如果 t 在$[0,1]$内，P 是 A、B 之间的线段；

如果 $t>0$，P 是射线。

```
struct Line{
```

```
    Point p1,p2;                        //线上的两个点
    Line(){}
    Line(Point p1,Point p2):p1(p1),p2(p2){}
    //根据一个点和倾斜角 angle 确定直线,0≤angle<pi
    Line(Point p,double angle){
        p1 = p;
        if(sgn(angle - pi/2) == 0){p2 = (p1 + Point(0,1));}
        else{p2 = (p1 + Point(1,tan(angle)));}
    }
    //ax + by + c = 0
    Line(double a,double b,double c){
        if(sgn(a) == 0){
            p1 = Point(0,-c/b);
            p2 = Point(1,-c/b);
        }
        else if(sgn(b) == 0){
            p1 = Point(-c/a,0);
            p2 = Point(-c/a,1);
        }
        else{
            p1 = Point(0,-c/b);
            p2 = Point(1,(-c-a)/b);
        }
    }
};
```

2. 线段的表示

可以用两个点表示线段,起点是 p_1,终点是 p_2。直接用直线的数据结构定义线段:

```
typedef Line Segment;
```

3. 点和直线的位置关系

在二维平面上,点和直线有 3 种位置关系,即点在直线左侧、在右侧、在直线上。用直线上的两点 p_1 和 p_2 与点 p 构成两个向量,用叉积的正负判断方向,就能得到位置关系。

```
int Point_line_relation(Point p, Line v){
    int c = sgn(Cross(p-v.p1,v.p2-v.p1));
    if(c < 0)return 1;                  //1: p 在 v 的左边
    if(c > 0)return 2;                  //2: p 在 v 的右边
    return 0;                           //0: p 在 v 上
}
```

4. 点和线段的位置关系

判断点 p 是否在线段 v 上,先用叉积判断是否共线,然后用点积看 p 和 v 的两个端点产生的角是否为钝角(实际上应该是 180°角)。

```
bool Point_on_seg(Point p, Line v){     //点和线段: 0 为点不在线段 v 上; 1 为点在线段 v 上
    return sgn(Cross(p-v.p1, v.p2-v.p1)) == 0 && sgn(Dot(p - v.p1,p- v.p2)) <= 0;
}
```

5. 点到直线的距离

已知点 p 和直线 $v(p_1,p_2)$，求 p 到 v 的距离。首先用叉积求 p、p_1、p_2 构成的平行四边形的面积，然后用面积除以平行四边形的底边长，也就是线段(p_1, p_2)的长度，就得到了平行四边形的高，即 p 点到直线的距离。

```
double Dis_point_line(Point p, Line v){
    return fabs(Cross(p - v.p1,v.p2 - v.p1))/Distance(v.p1,v.p2);
}
```

6. 点在直线上的投影

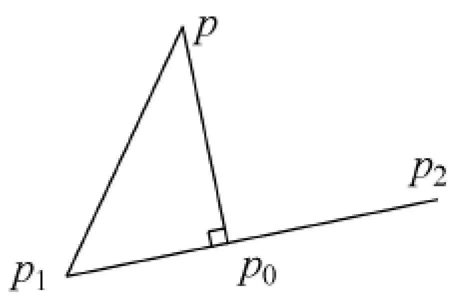


图 11.5　点在直线上的投影

已知直线上的两点 p_1、p_2 以及直线外的一点 p，求投影点 p_0，如图 11.5 所示。

令 $k=\frac{|p_0-p_1|}{|p_2-p_1|}$，即 k 是线段 p_0p_1 和 p_2p_1 长度的比值。因为 $p_0=p_1+k*(p_2-p_1)$，如果求得 k，就能得到 p_0。

根据点积的概念，有

$$(p-p_1)\cdot(p_2-p_1)=|p_2-p_1|*|p_0-p_1|$$

即 $|p_0-p_1|=\frac{(p-p_1)\cdot(p_2-p_1)}{|p_2-p_1|}$，代入得

$$k=\frac{|p_0-p_1|}{|p_2-p_1|}=\frac{(p-p_1)\cdot(p_2-p_1)}{|p_2-p_1|*|p_2-p_1|}$$

所以

$$p_0=p_1+k*(p_2-p_1)=p_1+\frac{(p-p_1)\cdot(p_2-p_1)}{|p_2-p_1|*|p_2-p_1|}*(p_2-p_1)$$

代码如下：

```
Point Point_line_proj(Point p, Line v){
    double k = Dot(v.p2 - v.p1,p - v.p1)/Len2(v.p2 - v.p1);
    return v.p1 + (v.p2 - v.p1) * k;
}
```

7. 点关于直线的对称点

求一个点 p 对一条直线 v 的镜像点。先求点 p 在直线上的投影 q，再求对称点 p'，如图 11.6 所示。

```
Point Point_line_symmetry(Point p, Line v){
    Point q = Point_line_proj(p,v);
    return Point(2 * q.x - p.x,2 * q.y - p.y);
}
```

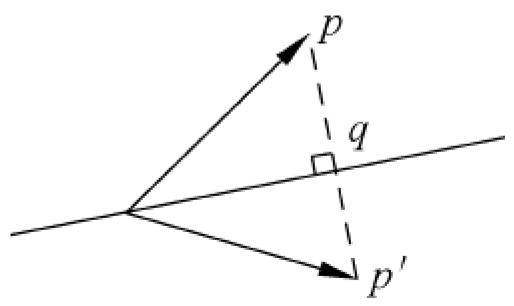


图 11.6　对称点

8. 点到线段的距离

对于点 p 到线段 AB 的距离，在以下 3 个距离中取最小值：从 p 出发对 AB 做垂线，如果交点在 AB 线段上，这个距离就是最小值；p 到 A 的距离；p 到 B 的距离。

```
double Dis_point_seg(Point p, Segment v){
```

```
    if(sgn(Dot(p - v.p1,v.p2 - v.p1))< 0 || sgn(Dot(p - v.p2,v.p1 - v.p2))< 0)
        return min(Distance(p,v.p1),Distance(p,v.p2));
    return Dis_point_line(p,v);             //点的投影在线段上
}
```

9. 两条直线的位置关系

```
int Line_relation(Line v1, Line v2){
    if(sgn(Cross(v1.p2 - v1.p1,v2.p2 - v2.p1)) == 0){
        if(Point_line_relation(v1.p1,v2) == 0) return 1;    //1:重合
        else return 0;                                      //0:平行
    }
    return 2;                                               //2:相交
}
```

10. 求两条直线的交点

对于两直线的交点，可以通过 $a_1x+b_1y+c_1=0$ 与 $a_2x+b_2y+c_2=0$ 联立方程求解。不过，借助叉积有更简单的方法。

图 11.7 中有 4 个点 A、B、C、D，组成两条直线 AB 和 CD，交点是 P。以下两个关系成立：

$\frac{|DP|}{|CP|}=\frac{S_{\triangle ABD}}{S_{\triangle ABC}}=\frac{\overrightarrow{AD}\times\overrightarrow{AB}}{\overrightarrow{AB}\times\overrightarrow{AC}}$，其中 $S_{\triangle ABD}$、$S_{\triangle ABC}$ 表示三角形的面积。

$\frac{|DP|}{|CP|}=\frac{x_D-x_P}{x_P-x_C}=\frac{y_D-y_P}{y_P-y_C}$，其中 x_D、y_D 等表示点的坐标。

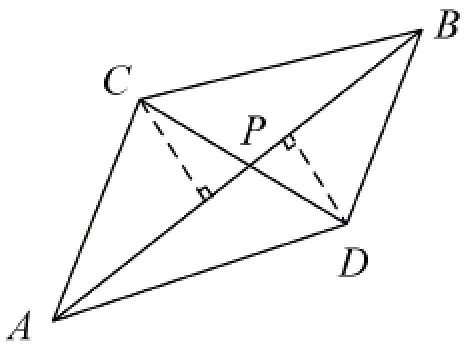


图 11.7 直线的交点

联立上面两个方程，得到交点 P 的坐标如下：

$$x_P=\frac{S_{\triangle ABD}\times x_C+S_{\triangle ABC}\times x_D}{S_{\triangle ABD}+S_{\triangle ABC}}$$

$$y_P=\frac{S_{\triangle ABD}\times y_C+S_{\triangle ABC}\times y_D}{S_{\triangle ABD}+S_{\triangle ABC}}$$

三角形的面积可以通过叉积求得：$S_{\triangle ABD}=\overrightarrow{AD}\times\overrightarrow{AB}$，$S_{\triangle ABC}=\overrightarrow{AB}\times\overrightarrow{AC}$。

程序如下：

```
Point Cross_point(Point a,Point b,Point c,Point d){     //Line1:ab, Line2:cd
   double s1  =  Cross(b - a,c - a);
   double s2  =  Cross(b - a,d - a);                    //叉积有正负
   return Point(c.x * s2 - d.x * s1,c.y * s2 - d.y * s1)/(s2 - s1);
}
```

注意：在 Cross_point()中要对(s2−s1)做除法，所以在调用 Cross_point()之前应该保证 s2−s1≠0，即直线 AB、CD 不共线，而且不平行。

11. 判断两个线段是否相交

这里仍然利用叉积有正负的特点。如果一条线段的两端在另一条线段的两侧，那么两个端点与另一线段产生的两个叉积正负相反，也就是说两个叉积相乘为负。如果两条线段互相满足这一点，那么就是相交的。

```
bool Cross_segment(Point a,Point b,Point c,Point d){    //Line1:ab;  Line2:cd
    double c1 = Cross(b - a,c - a),c2 = Cross(b - a,d - a);
    double d1 = Cross(d - c,a - c),d2 = Cross(d - c,b - c);
    return sgn(c1) * sgn(c2)< 0 && sgn(d1) * sgn(d2)< 0;   //1:相交; 0:不相交
}
```

12. 求两条线段的交点

先判断两条线段是否相交，若相交，问题转化成两条直线求交点。

11.1.4 多边形

1. 判断点在多边形内部

给定一个点 P 和一个多边形，判断 P 是否在多边形内部，有射线法和转角法两种方法。

射线法：从 P 引一条射线，穿过多边形，如果和多边形的边相交奇数次，说明 P 在外部；如果是偶数次，说明在内部。这种方法比较烦琐，很少使用。

转角法：把点 P 和多边形的每个点连接，逐个计算角度，绕多边形一周，看多边形相对于这个点总共转了多少度。如果是 360°，说明点在多边形内；如果是 0°，说明点在多边形外；如果是 180°，说明点在多边形边界上。但是，如果直接算角度，需要计算反三角函数，不仅速度慢，而且有精度问题。

下面的方法是转角法思想的另一种实现：以点 P 为起点引一条水平线，检查与多边形每条边的相交情况，例如沿着逆时针，检查 P 和每个边的相交情况，统计 P 穿过这些边的次数。见图 11.8 和图 11.9，检查以下 3 个参数：

$$c = \text{Cross}(P - j, i - j)$$
$$u = i.y - P.y$$
$$v = j.y - P.y$$

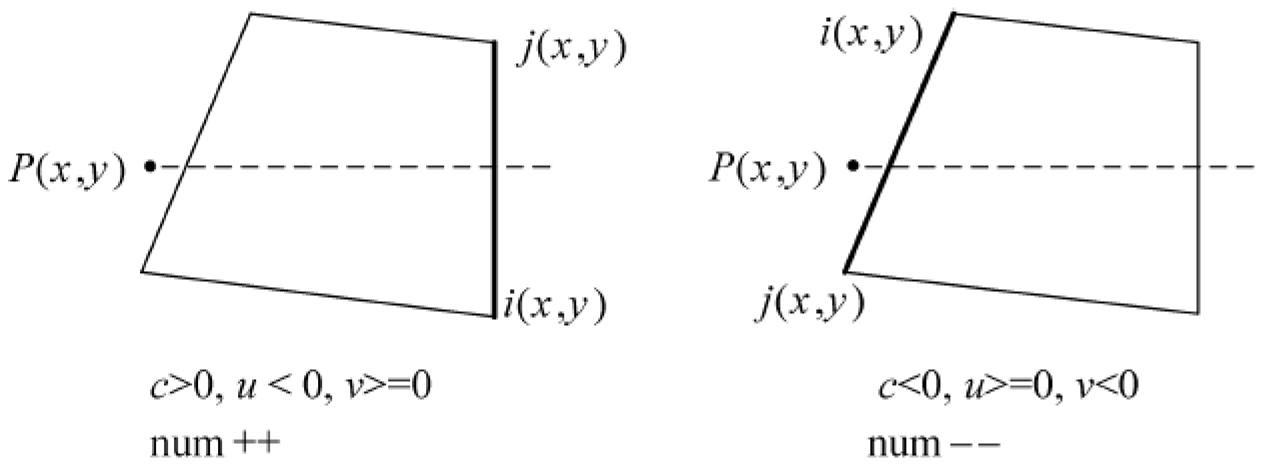


图 11.8　P 在多边形左侧

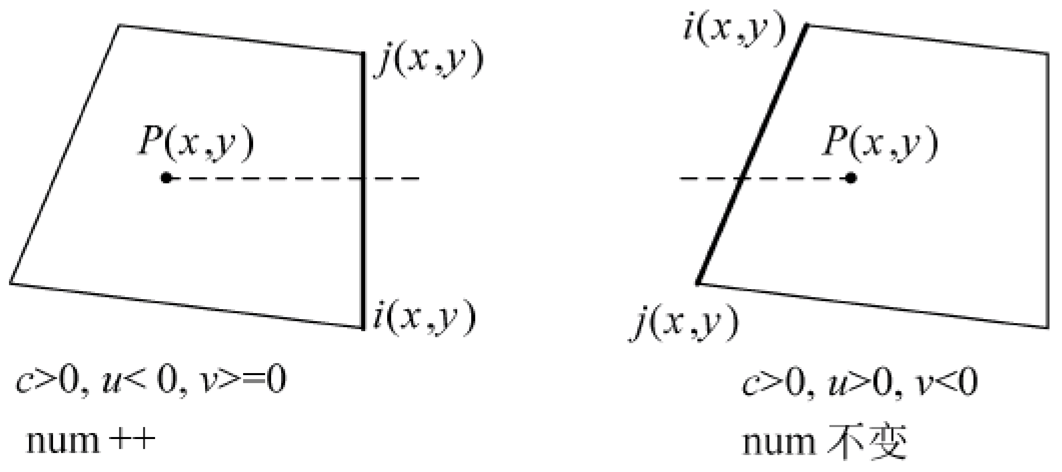


图 11.9　P 在多边形内部

叉积 c 用来检查 P 点在线段 ij 的左侧还是右侧，u、v 用来检查经过 P 的水平线是否穿过线段 ij。

用 num 计数：

```
if(c > 0 && u < 0 && v >= 0) num++;
if(c < 0 && u >= 0 && v < 0) num--;
```

当 num>0 时，P 在多边形内部。读者可以验证其他情况，例如 P 在多边形右侧、多边形是凹多边形，看上述判断是否成立。

下面是代码，注意多边形的形状是由各个顶点的排列顺序决定的。

```
int Point_in_polygon(Point pt,Point *p,int n){        //点 pt,多边形 Point *p
    for(int i = 0;i < n;i++){                         //3: 点在多边形的顶点上
        if(p[i] == pt)return 3;
    }
    for(int i = 0;i < n;i++){                         //2: 点在多边形的边上
        Line v = Line(p[i],p[(i + 1) % n]);
        if(Point_on_seg(pt,v)) return 2;
    }
    int num = 0;
    for(int i = 0;i < n;i++){
        int j = (i + 1) % n;
        int c = sgn(Cross(pt - p[j],p[i] - p[j]));
        int u = sgn(p[i].y - pt.y);
        int v = sgn(p[j].y - pt.y);
        if(c > 0 && u < 0 && v >= 0) num++;
        if(c < 0 && u >= 0 && v < 0) num--;
    }
    return num != 0;                                  //1: 点在内部; 0: 点在外部
}
```

2. 求多边形的面积

给定一个凸多边形，求它的面积。读者很容易想到，可以在凸多边形内部找一个点 P，然后以这个点为中心，与凸多边形的边结合，对多边形进行三角剖分，所有三角形的和就是凸多边形的面积。每个三角形的面积可以用叉积来求。

事实上，上述方法不仅可用于凸多边形，也适用于非凸多边形；而且点 P 并不需要在多边形内部，在任何位置都可以，例如以原点为 P，编程最简单。这是因为叉积是有正负的，它可以抵消多边形外部的面积，图 11.10 给出了各种情况。

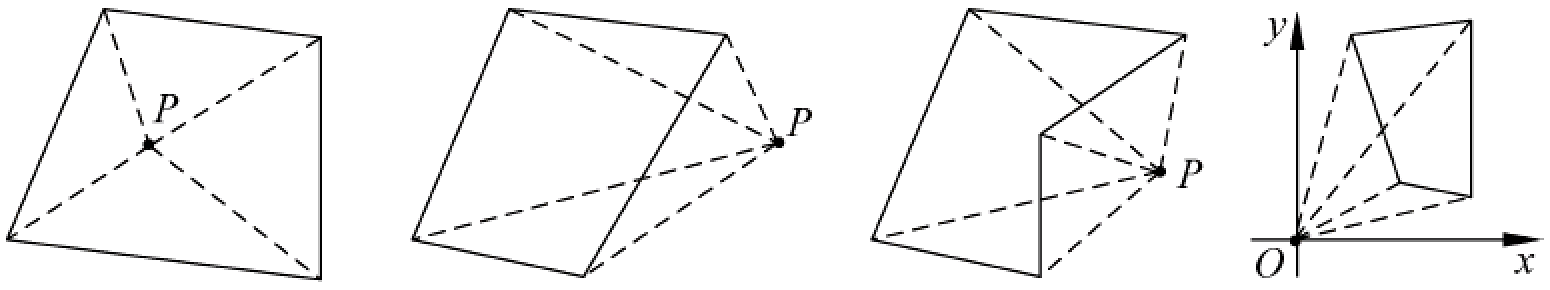


图 11.10 求任意多边形的面积

下面的程序以原点为中心点划分三角形，然后求多边形的面积。

```
double Polygon_area(Point *p, int n){                 //Point *p 表示多边形
    double area = 0;
```

```
    for(int i = 0;i < n;i++)
        area += Cross(p[i],p[(i + 1) % n]);
    return area/2;                              //面积有正负,这里不能简单地取绝对值
}
```

3. 求多边形的重心

将多边形三角剖分,算出每个三角形的重心,三角形的重心是 3 点坐标的平均值,然后对每个三角形的有向面积求加权平均。

下面用一个例题综合讲解前面一些模板的应用。代码中的 Polygon_center()是求多边形的重心。

hdu 1115 "Lifting the Stone"

给定一个 N 多边形,$3 \leqslant N \leqslant 1\ 000\ 000$,求重心。

代码如下:

```
#include <bits/stdc++.h>
struct Point{
    double x,y;
    Point(double X = 0,double Y = 0){x = X,y = Y;}
    Point operator + (Point B){return Point (x + B.x,y + B.y);}
    Point operator - (Point B){return Point (x - B.x,y - B.y);}
    Point operator * (double k){return Point (x * k,y * k);}
    Point operator / (double k){return Point (x/k,y/k);}
};
typedef Point Vector;
double Cross(Vector A,Vector B){return A.x * B.y - A.y * B.x;}
double Polygon_area(Point *p, int n){           //求多边形的面积
    double area = 0;
    for(int i = 0;i < n;i++)
        area += Cross(p[i],p[(i + 1) % n]);
    return area/2;                              //面积有正负,不能取绝对值
}
Point Polygon_center(Point *p, int n){          //求多边形的重心
    Point ans(0,0);
    if(Polygon_area(p,n) == 0) return ans;
    for(int i = 0;i < n;i++)
        ans = ans + (p[i] + p[(i + 1) % n]) * Cross(p[i],p[(i + 1) % n]);
    return ans/Polygon_area(p,n)/6;
}
int main(){
    int t,n,i;
    Point center;                               //重心的坐标
    Point p[100000];
    scanf("%d",&t);
    while(t--){
        scanf("%d",&n);
        for(i = 0;i < n;i++) scanf("%lf %lf",&p[i].x,&p[i].y);
```

```
        center = Polygon_center(p,n);
        printf("%.2f %.2f\n",center.x,center.y);  //注意这里输出用%f,不是用%lf
    }
    return 0;
}
```

【习题】

hdu 1558,几何+并查集。

11.1.5　凸包

凸包(Convex hull)是计算几何中的著名问题,有非常广泛的应用[①]。

凸包问题:给定一些点,求能把所有这些点包含在内的面积最小的多边形。可以想象有一个很大的橡皮箍,它把所有的点都箍在里面,在橡皮箍收紧之后,绕着最外围的点形成的多边形就是凸包。

求凸包的常用算法有两种,一是 Graham 扫描法,其复杂度是 $O(n\log_2 n)$;二是 Jarvis 步进法,其复杂度是 $O(nh)$,h 是凸包上的顶点数。这两种算法的基本思路是"旋转扫除",设定一个参照顶点,逐个旋转到其他所有顶点,并判断这些顶点是否在凸包上。

这里介绍 Graham 扫描法的变种——Andrew 算法,它更快、更稳定。算法做两次扫描,先从最左边的点沿"下凸包"扫描到最右边,再从最右边的点沿"上凸包"扫描到最左边,"上凸包"和"下凸包"合起来就是完整的凸包。

具体步骤如下:

(1) 把所有点按照横坐标 x 从小到大进行排序,如果 x 相同,按 y 从小到大排序,并删除重复的点,得到序列$\{p_0,p_1,p_2,\cdots,p_m\}$。

(2) 从左到右扫描所有点,求"下凸包"。p_0 一定在凸包上,它是凸包最左边的顶点,从 p_0 开始,依次检查$\{p_1,p_2,\cdots,p_m\}$,扩展出"下凸包"。判断的依据是:如果新点在凸包"前进"方向的左边,说明在"下凸包"上,把它加入到凸包;如果在右边,说明拐弯了,删除最近加入下凸包的点。继续这个过程,直到检查完所有点。拐弯方向用叉积判断即可。例如图 11.11 所示,在检查 p_4 时发现 p_4p_3 对 p_3p_2 是右拐弯的,说明 p_3 不在下凸包上(有可能在"上凸包"上,在步骤(3)中会判断);退回到 p_2,发现 p_4p_2 对 p_2p_1 也是右拐弯的,退回到 p_1。

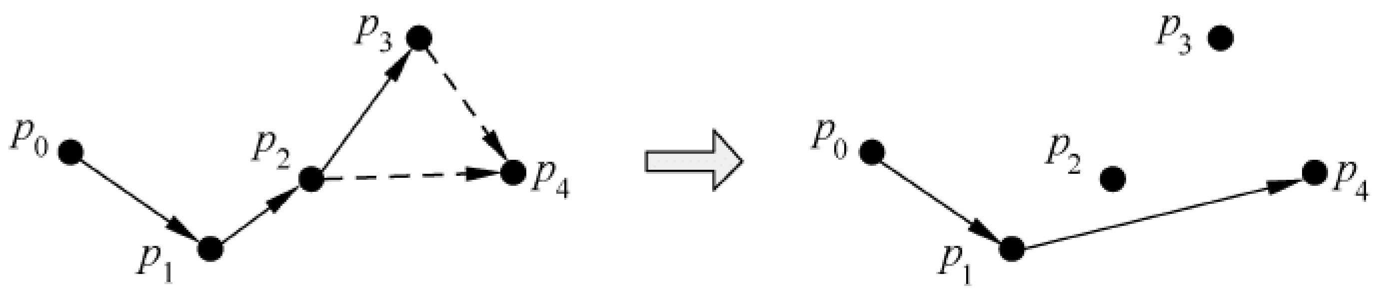


图 11.11　下凸包

① https://en.wikipedia.org/wiki/Convex_hull。

(3) 从右到左重新扫描所有点,求"上凸包"。和求"下凸包"的过程类似,最右边的点 p_m 一定在凸包上。

复杂度。算法先对点排序,复杂度是 $O(n\log_2 n)$①,然后扫描 $O(n)$次得到凸包。算法的总复杂度是 $O(n\log_2 n)$。

下面用一个例题讲解凸包模板的应用。代码中的 Convex_hull()是求凸包,注意其中用于去重的 unique()函数。

> **hdu 1392 "Surround the Trees"**
>
> 输入 n 个点,求凸包的周长。

代码如下:

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 104;
const double eps = 1e-8;
int sgn(double x){                    //判断 x 是否等于 0
    if(fabs(x) < eps)  return 0;
    else return x<0?-1:1;
}
struct Point{
    double x,y;
    Point(){}
    Point(double x, double y):x(x),y(y){}
    Point operator + (Point B){return Point(x+B.x,y+B.y);}
    Point operator - (Point B){return Point(x-B.x,y-B.y);}
    bool operator == (Point B){return sgn(x-B.x) == 0 && sgn(y-B.y) == 0;}
    bool operator < (Point B){        //用于 sort()排序
        return sgn(x-B.x)<0 || (sgn(x-B.x)==0 && sgn(y-B.y)<0);}
};
typedef Point Vector;
double Cross(Vector A,Vector B){return A.x*B.y - A.y*B.x;} //叉积
double Distance(Point A,Point B){return hypot(A.x-B.x,A.y-B.y);}
//Convex_hull()求凸包.凸包顶点放在 ch 中,返回值是凸包的顶点数
int Convex_hull(Point *p,int n,Point *ch){
    sort(p,p+n);                      //对点排序: 按 x 从小到大排序,如果 x 相同,按 y 排序
    n=unique(p,p+n)-p;                //去除重复点
    int v=0;
    //求下凸包.如果 p[i]是右拐弯的,这个点不在凸包上,往回退
    for(int i=0;i<n;i++){
        while(v>1 && sgn(Cross(ch[v-1]-ch[v-2],p[i]-ch[v-2]))<=0)
            v--;
        ch[v++]=p[i];
    }
    int j=v;
```

① 证明见《计算几何算法与应用(第 3 版)》,Mark de Berg 等著,邓俊辉译,清华大学出版社,8 页。

```
    //求上凸包
    for(int i = n - 2;i >= 0;i -- ){
        while(v > j && sgn(Cross(ch[v - 1] - ch[v - 2],p[i] - ch[v - 2]))<= 0)
            v -- ;
        ch[v++] = p[i];
    }
    if(n > 1) v -- ;
    return v;                           //返回值 v 是凸包的顶点数
}
int main(){
    int n;
    Point p[maxn],ch[maxn];             //输入点是 p[ ],凸包顶点放在 ch[ ]中
    while(scanf(" % d",&n) && n){
        for(int i = 0;i < n;i++) scanf(" % lf % lf",&p[i].x,&p[i].y);
        int v  =  Convex_hull(p,n,ch);  //返回凸包的顶点数 v
        double ans = 0;
        if(v == 1) ans = 0;
        else if(v == 2) ans = Distance(ch[0],ch[1]);
        else
            for(int i = 0;i < v;i++)    //计算凸包的周长
                ans += Distance(ch[i],ch[(i + 1) % v]);
        printf(" % .2f\n",ans);
    }
    return 0;
}
```

【习题】

hdu 6325 “Interstellar Travel”。

11.1.6 最近点对

平面最近点对问题：给定平面上的 n 个点，找出距离最近的两个点。

先考虑暴力法，即列出所有的点对，然后比较每一对的距离，找出其中最短的。n 个点有 $c(n,2)$ 种组合，复杂度是 $O(n^2)$。

最近点对的标准算法是分治法，复杂度是 $O(n\log_2 n)$。下面是思路：

划分。把点的集合 S 平均分成两个子集 S_1 和 S_2(按点的 x 坐标排序，并按 x 的大小分成两半)，然后每个子集再划分成更小的两个子集，递归这个过程，直到子集中只有一个点或两个点。

解决。在每个子集中递归地求最近点对。

合并。在求出子集 S_1 和 S_2 的最接近点对后，合并 S_1 和 S_2。合并时有以下两种情况：

(1) 集合 S 中的最近点对在子集 S_1 内部或者 S_2 内部，那么可以简单地直接合并 S_1 和 S_2。

(2) 这两个点一个在 S_1 中，一个在 S_2 中，不能简单合并。设 S_1 中的最短距离是 d_1，S_2 中的最短距离是 d_2，在 S_1 和 S_2 的中间点 p[mid]附近找到所有离它小于 d_1 和 d_2 的点(仍

然按 x 坐标值计算距离)，记录在点集 tmp_p[]中，这样那么最近点对就在这些点中。这样在这些点中找最近点对就行了。用分治法求最近点对应如图 11.12 所示。但是，仍然不能直接用暴力法列出点集 tmp_p[]中的所有点对，否则会 TLE。可以先按 y 坐标值对 tmp_p[]的点排序(这次不能按 x 坐标值排序，请思考为什么)，然后用剪枝把不符合条件的去掉。具体见下面例题的代码。

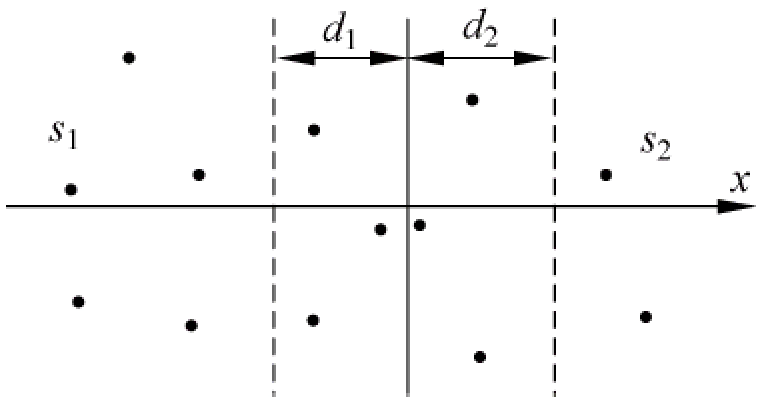


图 11.12　用分治法求最近点对

hdu 1007 "Quoit Design"

给定平面上的 n 个点，$2 \leqslant n \leqslant 100\ 000$。

找到最近点对，输出最近点对距离的一半。

下面是代码，注意其中分治法和剪枝的内容。程序比较简单，读者应该能自己写出来。

```
#include <bits/stdc++.h>
using namespace std;
const double eps = 1e-8;
const int MAXN = 100010;
const double INF = 1e20;
int sgn(double x){
    if(fabs(x) < eps)  return 0;
    else return x<0?-1:1;
}
struct Point{
    double x,y;
};
double Distance(Point A, Point B){return hypot(A.x-B.x,A.y-B.y);}
bool cmpxy(Point A,Point B){                          //排序：先对横坐标 x 排序，再对 y 排序
  return sgn(A.x-B.x)<0 || (sgn(A.x-B.x) == 0 && sgn(A.y-B.y)<0);
}
bool cmpy(Point A,Point B){return sgn(A.y-B.y)<0;}   //只对 y 坐标排序
Point p[MAXN],tmp_p[MAXN];
double Closest_Pair(int left,int right){
    double dis = INF;
    if(left == right) return dis;                    //只剩一个点
    if(left + 1 == right)                            //只剩两个点
        return Distance(p[left], p[right]);
    int mid = (left+right)/2;                        //分治
    double d1 = Closest_Pair(left,mid);              //求 s1 内的最近点对
    double d2 = Closest_Pair(mid+1,right);           //求 s2 内的最近点对
    dis = min(d1,d2);
    int k = 0;
    for(int i=left;i<=right;i++)                     //在 s1 和 s2 中间附近找可能的最小点对
        if(fabs(p[mid].x - p[i].x) <=dis)            //按 x 坐标来找
            tmp_p[k++] = p[i];
  sort(tmp_p,tmp_p+k,cmpy);                   //按 y 坐标排序，用于剪枝.这里不能按 x 坐标排序
```

```
    for(int i = 0;i < k;i++)
        for(int j = i + 1;j < k;j++){
            if(tmp_p[j].y - tmp_p[i].y >= dis)  break;    //剪枝
            dis = min(dis,Distance(tmp_p[i],tmp_p[j]));
        }
  return dis;                                             //返回最小距离
}
int main(){
    int n;
    while(~scanf("%d",&n) && n){
        for(int i = 0;i < n;i++) scanf("%lf%lf",&p[i].x,&p[i].y);
        sort(p,p + n,cmpxy);                              //先排序
        printf("%.2f\n",Closest_Pair(0,n - 1)/2);         //输出最短距离的一半
    }
    return 0;
}
```

【习题】

hdu 5721 “Palace”。

11.1.7 旋转卡壳

对于平面上的点集，可以用两条或更多平行线来“卡”住它们，从而解决很多问题。图 11.13 给出了一些应用场合。

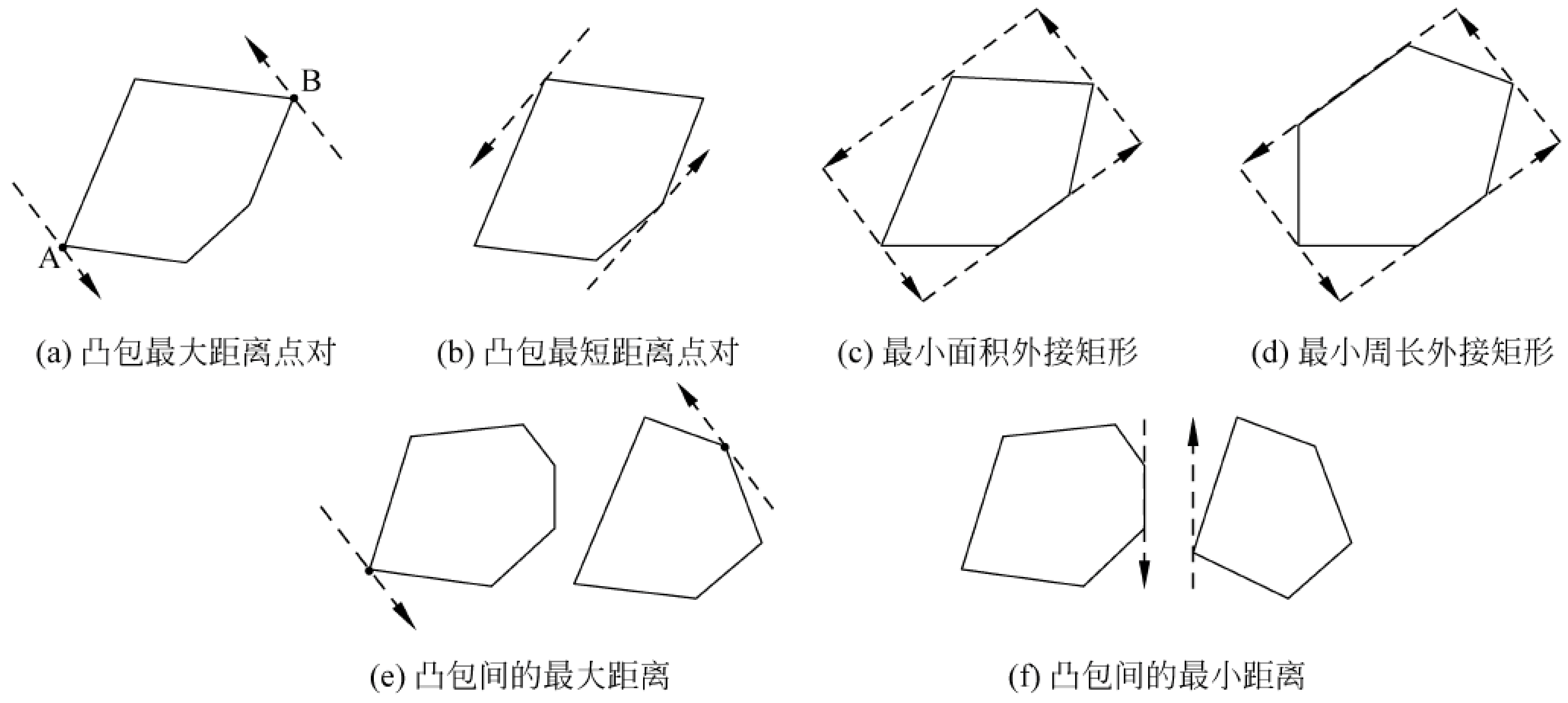


图 11.13 旋转卡壳的应用

两条平行线与凸包的交点称为对踵点对(antipodal pair)，例如图 11.13(a)中的 A、B 点。找对踵点对，可以使用被形象地称为旋转卡壳(rotating calipers)的方法。

旋转卡壳算法是这样操作的：

(1) 找初始的对踵点对和平行线。可以取 y 坐标最大和最小的两个点，经过这两个点做两条水平线，一条向左，一条向右。

(2) 同时逆时针旋转两条线,直到其中一条线与多边形的一条边重合。此时得到新的对踵点对。如果题目要求最大距离点对,可以计算新对踵点对的距离,并比较和更新。

(3) 重复(2),直到回到初始对踵点。

【习题】

hdu 2202,凸包+旋转卡壳。

hdu 2187/2823。

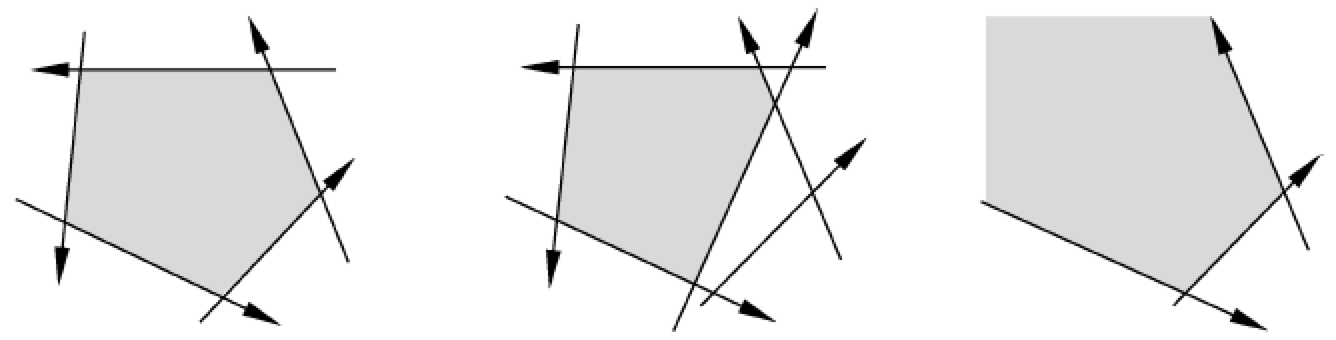
hdu 5251,凸包+旋转卡壳求最小矩形覆盖。

11.1.8 半平面交

半平面就是平面的一半。

一个半平面用一条有向直线来定义。一条直线把平面分成两部分,为区分这两部分,这条直线应该是有向的,可以定义它左侧的平面是它代表的半平面。

给定一些半平面,它们相交会围成一片区域,例如图 11.14 所示的情况。



(a) 围成一个凸多边形　(b) 新的凸多边形　(c) 不闭合的情况

图 11.14　半平面交

图 11.14(a)中的 5 个半平面围成了一个凸多边形。如果再添加一个穿过凸多边形的半平面,那么凸多边形会变成图 11.14(b)。半平面交也可能不会闭合成一个凸多边形,而是成为图 11.14(c)的无边界的情况。在编程时为方便处理,可以在合适的地方人为添加半平面,闭合为凸多边形。

半平面的交一定是凸多边形(可能不闭合),所以求解半平面交问题就是求解形成的凸多边形。

1. 半平面的定义

表示半平面的有向直线,定义如下:

```
struct Line{
    Point p;                                    //直线上的一个点
    Vector v;                                   //方向向量,它的左边是半平面
    double ang;                                 //极角,从 x 正半轴旋转到 v 的角度
    Line(){};
    Line(Point p, Vector v):p(p),v(v){ang = atan2(v.y, v.x);}
    bool operator < (Line &L){return ang < L.ang;}     //用于排序
};
```

2. 半平面交算法

半平面交有一个显而易见的算法,即增量法,描述如下:

(1) 初始凸包。先人为设定一个极大的矩形，作为初始凸多边形，它能把最后形成的凸多边形包含进来。

(2) 逐一添加半平面，更新凸多边形。例如添加半平面 K，如果它能切割当前的凸多边形，则保留 K 左边的点，删除它右边的点，并把 K 与原凸多边形的交点加入到新的凸多边形中。

增量法不太好，它的复杂度是 $O(n^2)$，即一共有 n 次切割，每次切割都是 $O(n)$ 的。在下一页的例题 hdu 2297 中，$0<n\leqslant 50\,000$，用增量法会 TLE。

下面介绍的算法，其复杂度为 $O(n\log_2 n)$。

思考半平面交最终形成的凸多边形，沿逆时针顺序看，它的边的极角(或者斜率)是单调递增的。那么，可以先按极角递增的顺序对半平面进行排序，然后逐个进行半平面交，最后就得到了凸多边形。在这个过程中，用一个双端队列记录构成凸多边形的半平面：队列的首部指向最早加入凸多边形的半平面，尾部指向新加入的半平面。

算法的具体步骤如下：

(1) 对所有半平面按极角排序。

(2) 初始时，加入第 1 个半平面，双端队列的首部和尾部都指向它。

(3) 逐个加入和处理半平面。图 11.15 演示了基本情况，原来半平面只有 1 和 2，加入半平面 3。注意，由于半平面已经排序，3 的极角比 1、2 大，所以有图 11.15 所示的 4 种情况。

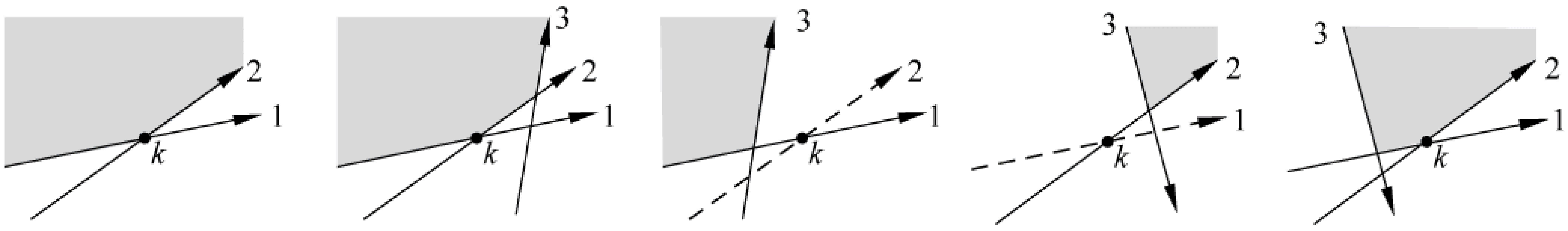
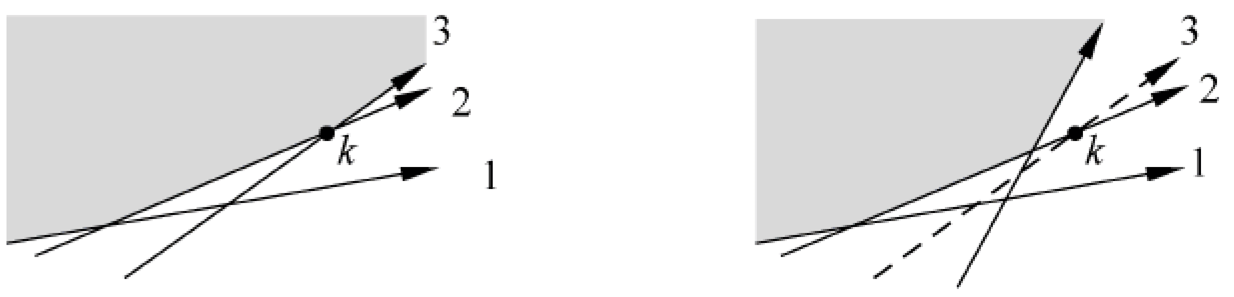


图 11.15　在半平面 1 和 2 上加入半平面 3 的 4 种情况

如果当前双端队列中不止有两个半平面，可以根据上面的讨论进行扩展。例如当前处理到半平面 L_i，有 4 种情况：L_i 可以直接加入队列；L_i 覆盖了原队尾；L_i 覆盖了原队首；L_i 不能加入到队列。下面讨论后面 3 种情况。

情况 1：L_i 覆盖原队尾。操作是：while 队尾的两个半平面的交点在 L_i 外面，那么删除队尾半平面。例如在图 11.16(a)中，队尾的两个半平面 L_2、L_3 的交点是 k。图(b)中新加入半平面 L_4，因为 k 在 L_4 的外面(点 k 在有向直线 L_4 的右边)，删除队尾的半平面 L_3。

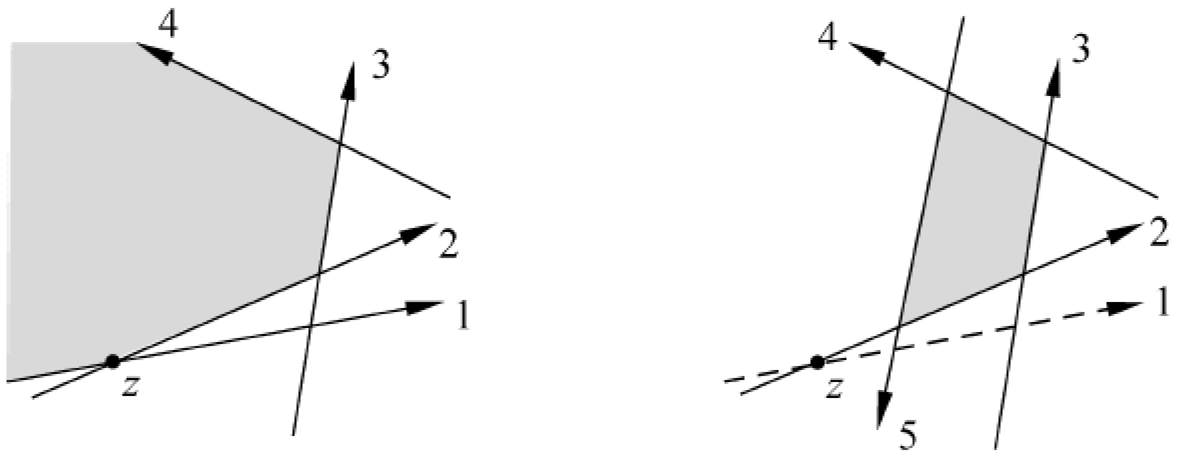


(a) 队尾的半平面交点k　　(b) k在L_4的外面，删除L_3

图 11.16　处理队尾

情况 2：L_i 覆盖原队首。操作是：while 队首的两个半平面的交点在 L_i 外面，那么删除队首的半平面。例如在图 11.17(a)中，队首 L_1、L_2 的两个半平面的交点是 z，图(b)中新加

入半平面 L_5，因为 z 在 L_5 的外面，删除队首的半平面 L_1。



(a) 队首半的平面交点 z　　(b) z 在 L_5 的外面，删除 L_1

图 11.17　处理队首

情况 3：L_i 不能加入到队列。例如图 11.18 所示的半平面 L_5，在步骤(3)中是合法的，但是它其实是无用的，不能加入到队列。判断条件是：尾部 L_4、L_5 的交点 r 在首部 L_1 的外面，则删除 L_5。

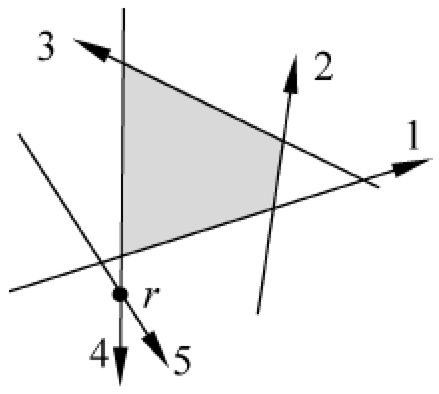


图 11.18　删除无用半平面 L_5

上述步骤的代码实现见下面的例题。

复杂度分析。排序，复杂度是 $O(n\log_2 n)$；逐个加入半平面，共检查 $O(n)$ 次，所以总复杂度是 $O(n\log_2 n)$。

3. 半平面的应用

下面的题目是很好的例子，它是半平面交的一个应用场景。

> **hdu 2297 “Run”**
>
> n 个人($0<n\leqslant 50\ 000$)在一条笔直的路上跑马拉松。设初始时每个人处于不同的位置，然后每个人都以自己的恒定速度不停地往前跑。
>
> 给定这 n 个人的初始位置和速度，问有多少人可能在某时刻成为第一？

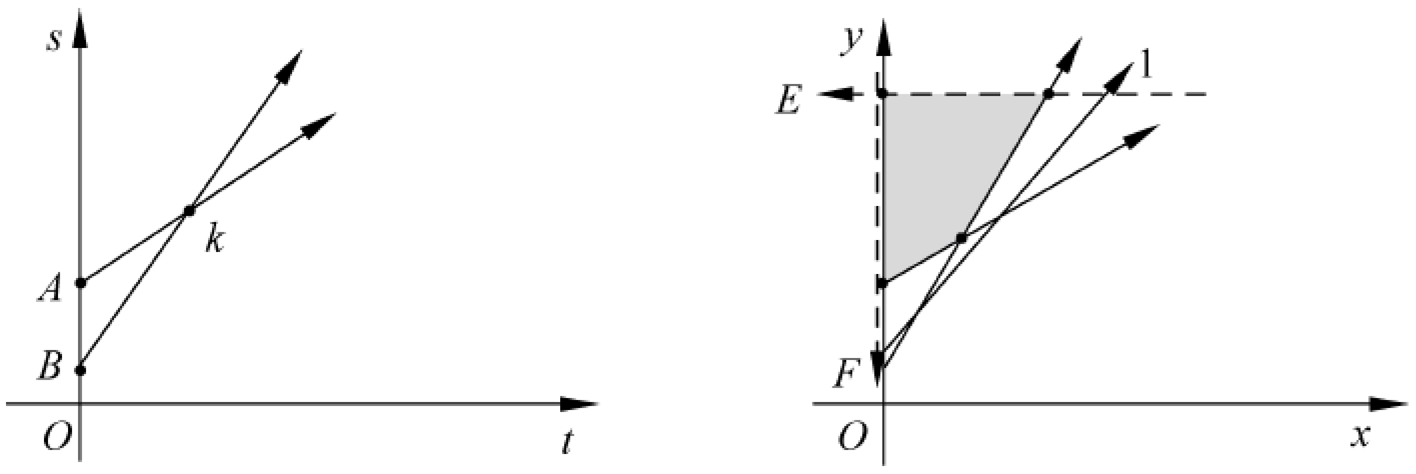
读者可以先思考：这一题如何建模为平面几何的半平面问题？

这一题实际上是半平面交的裸题，下面是建模过程。

以时间 t 为横轴，距离 s 为纵轴。设某人的初始位置在 A 点，从 A 出发画一条直线。他在某个时间段 Δt 内经过距离 Δs，两者的比值是直线的斜率，其物理意义正好是速度。他在某时刻 t' 的位置就是他在这条直线上的纵坐标 s'。这条直线代表了他的运动轨迹。运动轨迹始终位于第一象限。

图 11.19(a)中的两条直线是两个人 A 和 B 的运动轨迹，交叉点 k 是 B 追上 A 的点。

如果有 n 个人，那么就有 n 条直线在第一象限，见图(b)。相交的点是追上的点，但追上后不一定排第一，例如图中的线 1，它与其他线有两个交点，但都不是第一。只有凸面上的点才是题目要求的排名第一的点。另外，由于这些直线的半平面交不是一个完整的凸多边形，为方便编程，可以加两个半平面 E 和 F，形成闭合的凸多边形，其中 E 是 y 值无穷大的向左的水平线，F 是反向的 y 轴。图中阴影是半平面交形成的凸多边形，凸多边形的顶点数量去掉最上面的两个黑点，就是题目要求的排过第一名的数量。



(a) B追赶A　　(b) 半平面交

图 11.19　追赶问题

下面是 hdu 2297 的代码[①]。

```
#include<bits/stdc++.h>
using namespace std;
const double INF = 1e12;
const double pi = acos(-1.0);
const double eps = 1e-8;
int sgn(double x){
    if(fabs(x) < eps)  return 0;
    else return x<0?-1:1;
}
struct Point{
    double x,y;
    Point(){}
    Point(double x,double y):x(x),y(y){}
    Point operator + (Point B){return Point(x+B.x,y+B.y);}
    Point operator - (Point B){return Point(x-B.x,y-B.y);}
    Point operator * (double k){return Point(x*k,y*k);}
};
typedef Point Vector;
double Cross(Vector A,Vector B){return A.x*B.y - A.y*B.x;}  //叉积
struct Line{
    Point p;
    Vector v;
    double ang;
    Line(){};
    Line(Point p,Vector v):p(p),v(v){ang=atan2(v.y,v.x);}
    bool operator < (Line &L){return ang<L.ang;}            //用于极角排序
};
//点 p 在线 L 的左边,即点 p 在线 L 的外面
bool OnLeft(Line L,Point p){return sgn(Cross(L.v,p-L.p))>0;}
Point Cross_point(Line a,Line b){                            //两直线的交点
    Vector u=a.p-b.p;
    double t=Cross(b.v,u)/Cross(a.v,b.v);
    return a.p+a.v*t;
}
```

① 其中 HPI()的代码改编自《算法竞赛入门经典训练指南》,刘汝佳、陈锋著,清华大学出版社,278 页。

```
vector<Point> HPI(vector<Line> L){              //求半平面交,返回凸多边形
    int n = L.size();
    sort(L.begin(),L.end());                    //将所有半平面按照极角排序
    int first,last;                             //指向双端队列的第一个和最后一个元素
    vector<Point> p(n);                         //两个相邻半平面的交点
    vector<Line> q(n);                          //双端队列
    vector<Point> ans;                          //半平面交形成的凸包
    q[first = last = 0] = L[0];
    for(int i = 1;i<n;i++){
        //情况 1: 删除尾部的半平面
        while(first<last && !OnLeft(L[i], p[last-1])) last--;
        //情况 2: 删除首部的半平面
        while(first<last && !OnLeft(L[i], p[first]))  first++;
        q[++last] = L[i];                       //将当前的半平面加入双端队列的尾部
        //极角相同的两个半平面保留左边
        if(fabs(Cross(q[last].v,q[last-1].v)) < eps){
            last--;
            if(OnLeft(q[last],L[i].p)) q[last] = L[i];
        }
        //计算队列尾部的半平面交点
        if(first<last) p[last-1] = Cross_point(q[last-1],q[last]);
    }
    //情况 3: 删除队列尾部的无用半平面
    while(first<last && !OnLeft(q[first],p[last-1])) last--;
    if(last-first<=1) return ans;               //空集
    p[last] = Cross_point(q[last],q[first]);    //计算队列首尾部的交点
    for(int i = first;i<=last;i++)  ans.push_back(p[i]);   //复制
    return ans;                                 //返回凸多边形
}
int main(){
    int T,n;
    cin>>T;
    while(T--){
        cin>>n;
        vector<Line> L;
        //加一个半平面 F:反向 y 轴
        L.push_back(Line(Point(0,0),Vector(0,-1)));
        //加一个半平面 E:y 极大的向左的直线
        L.push_back(Line(Point(0,INF),Vector(-1,0)));
        while(n--){
            double a,b;
            scanf("%lf%lf",&a,&b);
            L.push_back(Line(Point(0,a),Vector(1,b)));
        }
        vector<Point> ans = HPI(L);             //得到凸多边形
        printf("%d\n",ans.size()-2);            //去掉人为加的两个点
    }
    return 0;
}
```

【习题】

hdu 4316,凸包+半平面交。
hdu 3982,半平面交。

11.2　圆

11.2.1　基本计算

1. 圆的定义

用圆心和半径表示圆。

```
struct Circle{
    Point c;                                    //圆心
    double r;                                   //半径
    Circle(){}
    Circle(Point c,double r):c®,r®{}
    Circle(double x,double y,double _r){c = Point(x,y);r = _r;}
};
```

2. 点和圆的关系

点和圆的关系根据点到圆心的距离判断。

```
int Point_circle_relation(Point p, Circle C){
    double dst = Distance(p,C.c);
    if(sgn(dst - C.r) < 0) return 0;            //0: 点在圆内
    if(sgn(dst - C.r) == 0) return 1;           //1: 点在圆上
    return 2;                                   //2: 点在圆外
}
```

3. 直线和圆的关系

直线和圆的关系根据圆心到直线的距离判断。

```
int Line_circle_relation(Line v,Circle C){
    double dst = Dis_point_line(C.c,v);
    if(sgn(dst - C.r) < 0) return 0;            //0: 直线和圆相交
    if(sgn(dst - C.r) == 0) return 1;           //1: 直线和圆相切
    return 2;                                   //2: 直线在圆外
}
```

4. 线段和圆的关系

线段和圆的关系根据圆心到线段的距离判断。

```
int Seg_circle_relation(Segment v,Circle C){
    double dst = Dis_point_seg(C.c,v);
```

```
    if(sgn(dst - C.r) < 0) return 0;                  //0: 线段在圆内
    if(sgn(dst - C.r)  == 0) return 1;                //1: 线段和圆相切
    return 2;                                         //2: 线段在圆外
}
```

5. 直线和圆的交点

求直线和圆的交点可以按图 11.20 所示，先求圆心 c 在直线上的投影 q，再求距离 d，然后根据 r 和 d 求出长度 k，最后求出两个交点 $p_a = q + n * k$、$p_b = q - n * k$，其中 k 是直线的单位向量。

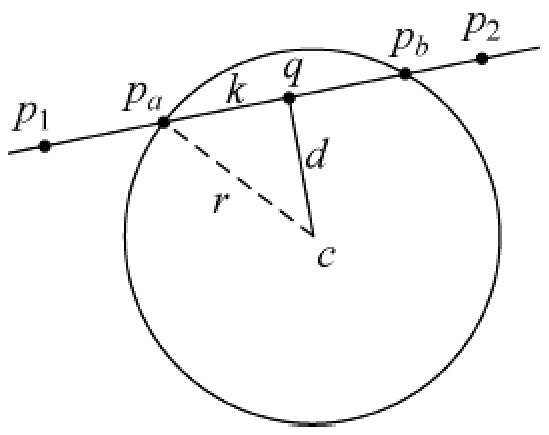


图 11.20　直线和圆的交点

```
//pa、pb 是交点.返回值是交点的个数
int Line_cross_circle(Line v,Circle C,Point &pa,Point &pb){
    if(Line_circle_relation(v, C) == 2)  return 0;    //无交点
    Point q  =  Point_line_proj(C.c,v);               //圆心在直线上的投影点
    double d  =  Dis_point_line(C.c,v);               //圆心到直线的距离
    double k  =  sqrt(C.r * C.r - d * d);
    if(sgn(k)  ==  0){                                //一个交点,直线和圆相切
        pa  =  q;pb  =  q;return 1;
    }
    Point n = (v.p2 - v.p1)/ Len(v.p2 - v.p1);        //单位向量
    pa  =  q  +  n * k;   pb  =  q  -  n * k;
    return 2;                                         //两个交点
}
```

6. 模板的使用

下面用 hdu 5572 题演示点、线、圆的几何模板的使用，如图 11.21 所示。这一题出自 2015 年 ACM-ICPC 区域赛上海赛区的现场赛，题目的详细说明见 12.2.4 节。

hdu 5572 “An Easy Physics Problem”

在一个无限光滑的桌面上有一个固定的大圆柱体，还有一个体积忽略不计的小球。开始时球静止于 A 点，给它一个初始速度和方向，如果球撞到圆柱体，它会弹回，没有能量损失。经过一段时间，小球是否会经过 B 点？

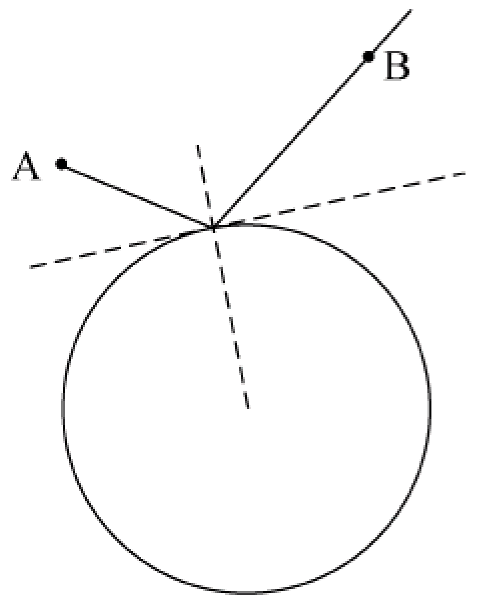


图 11.21　hdu 5572 题图示

这是一道中等题，考核参赛人员对几何基本模板的使用。该题目的逻辑很简单，但是综合性较强，它涉及的计算几何知识有直线的表示、圆的表示、点在直线上的投影、点到直线的距离、点对于直线的镜像点、线段和圆的关系、直线和圆的交点等。

下面的代码完全套用了前面给出的模板。

```
#include<bits/stdc++.h>
using namespace std;
const double eps = 1e-8;                //本题如果设定 eps = 1e-10，会 Wrong Answer
int sgn(double x){                      //判断 x 是否等于 0
    if(fabs(x) < eps)  return 0;
    else return x<0?-1:1;
}
struct Point{                           //定义点及其基本运算
    double x,y;
    Point(){}
    Point(double x,double y):x(x),y(y){}
    Point operator + (Point B){return Point(x+B.x,y+B.y);}
    Point operator - (Point B){return Point(x-B.x,y-B.y);}
    Point operator * (double k){return Point(x*k,y*k);}
    Point operator / (double k){return Point(x/k,y/k);}
};
typedef Point Vector;                   //定义向量
double Dot(Vector A,Vector B){return A.x*B.x + A.y*B.y;}    //点积
double Len(Vector A){return sqrt(Dot(A,A));}  //向量的长度
double Len2(Vector A){return Dot(A,A);}       //向量长度的平方
double Cross(Vector A,Vector B){return A.x*B.y - A.y*B.x;}  //叉积
double Distance(Point A, Point B){return hypot(A.x-B.x,A.y-B.y);}
struct Line{
    Point p1,p2;
    Line(){}
    Line(Point p1,Point p2):p1(p1),p2(p2){}
};
typedef Line Segment;                   //定义线段，两端点是 p1、p2
int Point_line_relation(Point p,Line v){
    int c = sgn(Cross(p-v.p1,v.p2-v.p1));
    if(c<0)return 1;                    //1: p 在 v 的左边
    if(c>0)return 2;                    //2: p 在 v 的右边
    return 0;                           //0: p 在 v 上
}
double Dis_point_line(Point p, Line v){       //点到直线的距离
    return fabs(Cross(p-v.p1,v.p2-v.p1))/Distance(v.p1,v.p2);
}
//点到线段的距离
double Dis_point_seg(Point p, Segment v){
    if(sgn(Dot(p-v.p1,v.p2-v.p1))<0 || sgn(Dot(p-v.p2,v.p1-v.p2))<0)
        return min(Distance(p,v.p1),Distance(p,v.p2));
    return Dis_point_line(p,v);
}
//点在直线上的投影
```

```
Point Point_line_proj(Point p, Line v){
    double k = Dot(v.p2 - v.p1,p - v.p1)/Len2(v.p2 - v.p1);
    return v.p1 + (v.p2 - v.p1) * k;
}
//点 p 对直线 v 的对称点
Point Point_line_symmetry(Point p, Line v){
    Point q = Point_line_proj(p,v);
    return Point(2 * q.x - p.x,2 * q.y - p.y);
}
struct Circle{
    Point c;                                          //圆心
    double r;                                         //半径
    Circle(){}
    Circle(Point c,double r):c(c),r(r){}
    Circle(double x,double y,double _r){c = Point(x,y);r = _r;}
};
//线段和圆的关系: 0 为线段在圆内,1 为线段和圆相切,2 为线段在圆外
int Seg_circle_relation(Segment v,Circle C){
    double dst = Dis_point_seg(C.c,v);
    if(sgn(dst - C.r) < 0) return 0;
    if(sgn(dst - C.r) == 0) return 1;
    return 2;
}
//直线和圆的关系: 0 为直线在圆内,1 为直线和圆相切,2 为直线在圆外
int Line_circle_relation(Line v,Circle C){
    double dst = Dis_point_line(C.c,v);
    if(sgn(dst - C.r) < 0) return 0;
    if(sgn(dst - C.r) == 0) return 1;
    return 2;
}
//直线和圆的交点,pa、pb 是交点.返回值是交点的个数
int Line_cross_circle(Line v,Circle C,Point &pa,Point &pb){
    if(Line_circle_relation(v, C) == 2) return 0;     //无交点
    Point q = Point_line_proj(C.c,v);                 //圆心在直线上的投影点
    double d = Dis_point_line(C.c,v);                 //圆心到直线的距离
    double k = sqrt(C.r * C.r - d * d);
    if(sgn(k) == 0){                                  //一个交点,直线和圆相切
        pa = q;  pb = q;  return 1;
    }
    Point n = (v.p2 - v.p1)/ Len(v.p2 - v.p1);        //单位向量
    pa = q + n * k;
    pb = q - n * k;
    return 2;                                         //两个交点
}
int main() {
    int T; scanf("%d", &T);
    for (int cas = 1; cas <= T; cas++) {
        Circle O; Point A,B,V;
        scanf("%lf%lf%lf", &O.c.x, &O.c.y, &O.r);
        scanf("%lf%lf%lf%lf", &A.x, &A.y, &V.x, &V.y);
        scanf("%lf%lf", &B.x, &B.y);
```

```
    Line l(A, A + V);                               //射线
    Line t(A, B);
        //情况 1: 直线和圆不相交,而且直线经过点
    if(Point_line_relation(B,l) == 0
        && Seg_circle_relation(t,O)>= 1 && sgn(Cross(B - A,V)) == 0)
        printf("Case # %d: Yes\n", cas);
    else{
        Point pa, pb;                               //直线和圆的交点
        //情况 2: 直线和圆相切,不经过点
        if(Line_cross_circle(l,O,pa,pb) != 2)
            printf("Case # %d: No\n",cas);
        //情况 3: 直线和圆相交
        else{
            Point cut;                              //直线和圆的碰撞点
            if(Distance(pa,A) > Distance(pb,A)) cut = pb;
            else cut = pa;
            Line mid(cut, O.c);                     //圆心到碰撞点的直线
            Point en = Point_line_symmetry(A,mid);  //镜像点
            Line light(cut, en);                    //反射线
            if(Distance(light.p2,B) > Distance(light.p1,B))
                swap(light.p1, light.p2);
            if(sgn(Cross(light.p2 - light.p1,
                      Point(B.x - cut.x,B.y - cut.y))) == 0)
            printf("Case # %d: Yes\n", cas);
            else
                printf("Case # %d: No\n", cas);
        }
    }
}
return 0;
}
```

11.2.2　最小圆覆盖

最小圆覆盖问题：给定 n 个点的平面坐标，求一个半径最小的圆，把 n 个点全部包围，部分点在圆上。

常见的算法有两种，即几何算法和模拟退火算法。

1. 几何算法

这个最小圆可以由 n 个点中的两个点或 3 个点确定。由两点定圆时，圆心是线段 AB 的中点，半径是 AB 长度的一半，其他点都在这个圆内；如果两点不足以包围所有点，就需要三点定圆，此时圆心是 A、B、C 这 3 个点组成的三角形的外心，如图 11.22 所示。

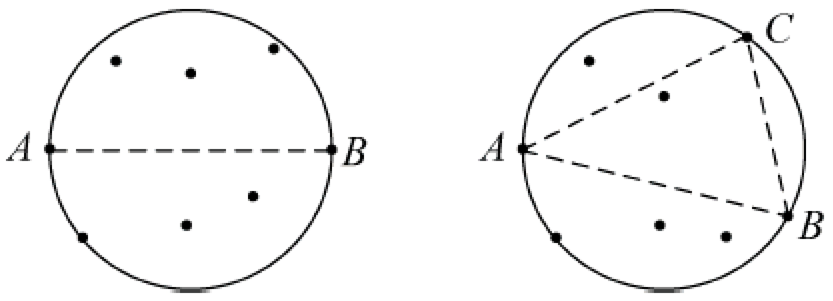


图 11.22　两点定圆或三点定圆

最小覆盖圆的获得就是寻找能两点定圆或三点定圆的那几个点。

一般用增量法求最小圆覆盖。算法从一个点开始，每次加入一个新的点，则更新最小圆，直到扩展到全部 n 个点。设前 i 个点的最小覆盖圆是 C_i，过程如下：

(1) 加第 1 个点 p_1。C_1 的圆心就是 p_1，半径为 0。

(2) 加第 2 个点 p_2。新的 C_2 的圆心是线段 p_1p_2 的中心，半径为两点距离的一半。这一步操作是两点定圆。

(3) 加第 3 个点 p_3。有两种情况：p_3 在 C_2 的内部或圆周上，不影响原来的最小圆，忽略 p_3；p_3 在 C_2 的外部，此时 C_2 已不能覆盖所有 3 个点，需要更新。下面讨论 p_3 在 C_2 外部的情况。因为 p_3 一定在新的 C_3 上，现在的任务转换为在 p_1、p_2 中找一个点或两个点，与 p_3 一起两点定圆或三点定圆。重新定圆的过程相当于回到第(1)步，把 p_3 作为第 1 个点加入，然后再加入 p_1、p_2。

(4) 加第 4 个点 p_4。分析和步骤(3)类似，为加强理解，这里重复说明一次。如果 p_4 在 C_3 的内部或圆周上，忽略它。如果在 C_3 的外部，那么需要求新的最小圆，此时 p_4 肯定在新的 C_4 的圆周上。任务转换为在 p_1、p_2、p_3 中找一个点或两个点，与 p_4 一起构成最小圆。先检查能不能找到一个点，用两点定圆；如果两点不够，就找到第 3 个点，用三点定圆。重新定圆的过程和前 3 个步骤类似，即把 p_4 作为第 1 个点加入，然后加入 p_1、p_2、p_3。

(5) 持续进行下去，直到加完所有点。

算法的思路概括如下：

假设已经求得前 $i-1$ 个点的 C_{i-1}，现在加入第 i 个点，有两种情况。

(1) i 在 C_{i-1} 的内部或圆周上，忽略 i。

(2) i 在 C_{i-1} 的外部，需要求新的 C_i。首先，i 肯定在 C_i 上，然后重新把前面的 $i-1$ 个点依次加入，根据两点定圆或者三点定圆重新构造最小圆。

几何算法的复杂度分析。在下面的例题中给出了模板代码。其中有 3 层 for 循环，看起来似乎是 $O(n^3)$。不过，如果点的分布是随机的，用概率进行分析可以得出程序的复杂度是接近 $O(n)$ 的。在下面的代码中，用 random_shuffle()函数进行随机打乱。

例如，如果前两个点 p_1 和 p_2 恰好就是最后的两点定圆，那么其他的所有点都只需要检查一次是否在 C_2 内就行了，程序在第一层 for 就结束了。对于算法复杂度的详细证明，请读者查阅有关资料。

下面的例题是最小圆覆盖的裸题。

hdu 3007 “Buried memory”

输入 n 个点的坐标，$n<500$，求最小圆覆盖。

代码如下：

```
#include <bits/stdc++.h>
using namespace std;
#define eps 1e-8
const int maxn = 505;
int sgn(double x){
    if(fabs(x) < eps)  return 0;
    else return x < 0? -1:1;
```

```
}
struct Point{
    double x, y;
};
double Distance(Point A, Point B){return hypot(A.x - B.x,A.y - B.y);}
//求三角形 abc 的外接圆的圆心
Point circle_center(const Point a, const Point b, const Point c){
    Point center;
    double a1 = b.x - a.x, b1 = b.y - a.y, c1 = (a1 * a1 + b1 * b1)/2;
    double a2 = c.x - a.x, b2 = c.y - a.y, c2 = (a2 * a2 + b2 * b2)/2;
    double d = a1 * b2 - a2 * b1;
    center.x = a.x + (c1 * b2 - c2 * b1)/d;
    center.y = a.y + (a1 * c2 - a2 * c1)/d;
    return center;
}
//求最小覆盖圆,返回圆心 c、半径 r:
void min_cover_circle(Point *p, int n, Point &c, double &r){
    random_shuffle(p, p + n);                   //随机函数,打乱所有点.这一步很重要
    c = p[0]; r = 0;                            //从第 1 个点 p0 开始.圆心为 p0,半径为 0
    for(int i = 1;i < n;i++)                    //扩展所有点
        if(sgn(Distance(p[i],c) - r)> 0){       //点 pi 在圆的外部
            c = p[i]; r = 0;                    //重新设置圆心为 pi,半径为 0
            for(int j = 0;j < i;j++)            //重新检查前面所有的点
                if(sgn(Distance(p[j],c) - r)> 0){   //两点定圆
                    c.x = (p[i].x + p[j].x)/2;
                    c.y = (p[i].y + p[j].y)/2;
                    r = Distance(p[j],c);
                    for(int k = 0;k < j;k++)
                        if (sgn(Distance(p[k],c) - r)> 0){  //两点不能定圆就三点定圆
                            c = circle_center(p[i],p[j],p[k]);
                            r = Distance(p[i], c);
                        }
                }
        }
}
int main(){
    int n;                                      //点的个数
    Point p[maxn];                              //输入点
    Point c; double r;                          //最小覆盖圆的圆心和半径
    while(~scanf("%d",&n) && n){
        for(int i = 0;i < n;i++) scanf("%lf %lf",&p[i].x,&p[i].y);
        min_cover_circle(p,n,c,r);
        printf("%.2f %.2f %.2f\n",c.x,c.y,r);
    }
    return 0;
}
```

2. 模拟退火算法

如果题目的数据规模不大,最小圆覆盖还可以用模拟退火算法实现,请读者先回顾第 6

章的“6.1.4 模拟退火”。用模拟退火求最小圆,不断迭代(降温);在每次迭代时,找到能覆盖到所有点的一个圆;在多次迭代中,逐步逼近最后要求的圆心和半径。

下面的函数 min_cover_circle()是模拟退火程序,用它替换上面的同名函数即可。

```
void min_cover_circle(Point *p, int n, Point &c, double &r){
    double T = 100.0;                       //初始温度
    double delta = 0.98;                    //降温系数
    c = p[0];
    int pos;
    while (T > eps){                        //eps 是终止温度
        pos = 0; r = 0;                     //初始: p[0]是圆心,半径是 0
        for(int i = 0; i <= n - 1; i++)     //找距圆心最远的点
            if (Distance(c, p[i]) > r){
                r = Distance(c, p[i]);      //距圆心最远的点肯定在圆周上
                pos = i;
            }
        c.x += (p[pos].x - c.x) / r * T;    //逼近最后的解
        c.y += (p[pos].y - c.y) / r * T;
        T *= delta;
    }
}
```

模拟退火的程序很简单,不过需要仔细选择初始温度 T、降温系数 delta、终止温度 eps 等,程序的复杂度也和它们有关。在本题中,模拟退火算法的复杂度远高于几何算法。OJ 返回的 AC 时间,几何算法是 46ms,模拟退火是 670ms。

视频讲解

【习题】

hdu 2215,最小圆覆盖。

11.3 三维几何

11.3.1 三维点和向量

1. 点和向量

在三维几何中,点和向量的表示和二维几何是类似的。同样也可以定义三维空间的运算。

```
struct Point3{                              //三维点
    double x,y,z;
    Point3(){}
    Point3(double x,double y,double z):x(x),y(y),z(z){}
    Point3 operator + (Point3 B){return Point3(x+B.x,y+B.y,z+B.z);}
    Point3 operator - (Point3 B){return Point3(x-B.x,y-B.y,z-B.z);}
    Point3 operator * (double k){return Point3(x*k,y*k,z*k);}
```

```
    Point3 operator / (double k){return Point3(x/k,y/k,z/k);}
    bool operator == (Point3 B){
        return sgn(x - B.x) == 0 && sgn(y - B.y) == 0 && sgn(z - B.z) == 0;}
};
typedef Point3 Vector3;                      //三维向量
```

点和点的距离[1]:

```
double Distance(Vector3 A,Vector3 B){
    return sqrt((A.x - B.x) * (A.x - B.x) +
                (A.y - B.y) * (A.y - B.y) +
                (A.z - B.z) * (A.z - B.z));    }
```

2. 线和线段

和二维一样,三维的直线和线段也用两点定义。

```
struct Line3{
    Point3 p1,p2;
    Line3(){}
    Line3(Point3 p1,Point3 p2):p1(p1),p2(p2){}
};
typedef Line3 Segment3;                      //定义线段,两端点是 Point p1,p2
```

11.3.2　三维点积

1. 点积

三维点积的定义和二维的类似,定义如下:

$$\boldsymbol{A}\cdot\boldsymbol{B}=|\boldsymbol{A}||\boldsymbol{B}|\cos\theta$$

求向量 $\boldsymbol{A}$、$\boldsymbol{B}$ 点积的代码如下:

```
double Dot(Vector3 A,Vector3 B){return A.x * B.x + A.y * B.y + A.z * B.z;}
```

2. 点积的基本应用

和二维点积一样,三维点积有以下基本应用:

1) 判断向量 $\boldsymbol{A}$ 与 $\boldsymbol{B}$ 的夹角是钝角还是锐角

点积有正负,利用正负号可以判断向量的夹角:

若 $\mathrm{dot}(\boldsymbol{A},\boldsymbol{B})>0$,$\boldsymbol{A}$ 与 $\boldsymbol{B}$ 的夹角为锐角;

若 $\mathrm{dot}(\boldsymbol{A},\boldsymbol{B})<0$,$\boldsymbol{A}$ 与 $\boldsymbol{B}$ 的夹角为钝角;

若 $\mathrm{dot}(\boldsymbol{A},\boldsymbol{B})=0$,$\boldsymbol{A}$ 与 $\boldsymbol{B}$ 的夹角为直角。

2) 求向量 $\boldsymbol{A}$ 的长度

```
double Len(Vector3 A){return sqrt(Dot(A, A));}
```

或者是求长度的平方,避免开方运算:

```
double Len2(Vector3 A){return Dot(A, A);}
```

---

① 读者可能注意到,本章给出的三维函数和二维函数很多是重名的,例如这里的 Distance()。C++ 允许函数重载,所以即使在同一个程序中用同名来定义不同的函数也是允许的。而且建议重载函数,这样做可以简化编程。

3）求向量 **A** 与 **B** 的夹角大小

```
double Angle(Vector3 A,Vector3 B){return acos(Dot(A,B)/Len(A)/Len(B));}
```

11.3.3 三维叉积

二维叉积是一个带正负的数值，而三维叉积是一个向量。可以把三维向量 **A**、**B** 的叉积看成垂直于 **A** 和 **B** 的向量，如图 11.23 所示，其方向符合"右手定则"。

三维叉积的计算和二维叉积相似，不同的是计算后返回一个向量：

```
Vector3 Cross(Vector3 A,Vector3 B){
    return Point3(A.y * B.z - A.z * B.y, A.z * B.x - A.x * B.z, A.x * B.y -
A.y * B.x);
}
```

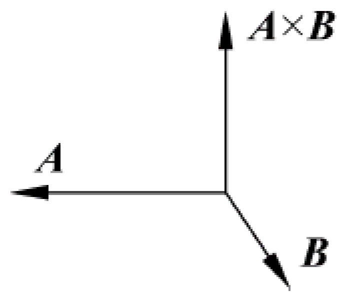


图 11.23　三维叉积

1. 三角形面积

三维的三角形面积计算和二维的相似，也是有向面积。先求三维叉积，然后取叉积的长度值。

```
//三角形面积的两倍
double Area2(Point3 A,Point3 B,Point3 C){return Len(Cross(B - A, C - A));}
```

判断点 p 是否在三角形 ABC 内，可以用 Area2()来计算。如果点 p 在三角形内部，那么用点 p 对三角形 ABC 进行三角剖分，形成的 3 个三角形的面积和与直接算 ABC 的面积，两者应该相等：

```
Dcmp(Area2(p,A,B) + Area2(p,B,C) + Area2(p,C,A), Area2(A,B,C)) == 0
```

2. 点和线的有关问题

点到直线的距离、点是否在直线上、点到线段的距离、点在直线上的投影等问题的代码和二维几何相似，见本章 11.4 节的几何模板。

3. 平面

用 3 个点可以确定一个平面。

```
struct Plane{
    Point3 p1,p2,p3;                        //平面上的 3 个点
    Plane(){}
    Plane(Point3 p1,Point3 p2,Point3 p3):p1(p1),p2(p2),p3(p3){}
};
```

4. 平面法向量

平面法向量是垂直于平面的向量，在平面问题中非常重要。它用叉积的概念计算即可，代码如下：

```
Point3 Pvec(Point3 A, Point3 B, Point3 C){return Cross(B - A,C - A);}
```

或者：

```
Point3 Pvec(Plane f){return Cross(f.p2 - f.p1,f.p3 - f.p1);}
```

5. 平面的有关问题

四点共平面、两平面平行、两平面垂直等问题的代码见本章11.4节的几何模板。

6. 直线和平面的交点

直线和平面有3种关系，即直线在平面上、直线和平面平行、直线和平面有交点。

一个平面，可以用平面 f 上的一点 $f.p1$ 以及平面的法向量 v 来决定。直线 u 用两点 $u.p1$ 和 $u.p2$ 决定。

下面的函数计算直线与平面的交点，交点是 p，函数的返回值是交点的个数。

```
int Line_cross_plane(Line3 u,Plane f,Point3 &p){
    Point3 v = Pvec(f);                       //平面的法向量
    double x = Dot(v, u.p2 - f.p1);
    double y = Dot(v, u.p1 - f.p1);
    double d = x - y;
    if(sgn(x) == 0 && sgn(y) == 0) return -1;//-1: v在f上
    if(sgn(d) == 0) return 0;                 //0: v与f平行
    p = ((u.p1 * x) - (u.p2 * y))/d;          //1: v与f相交
    return 1;
}
```

下面解释代码的正确性。

代码中的 v 是平面的法向量，它不一定是单位法向量，不过这里把它看成是单位法向量，不影响后续推理的正确性。$x=\mathrm{Dot}(v,u.p2-f.p1)$ 是 $u.p2$ 到平面 f 的距离，$y=\mathrm{Dot}(v,u.p1-f.p1)$ 是 $u.p1$ 到平面的距离。如果 $x=y=0$，说明直线在平面上；如果 $x=y\neq0$，即直线上的两点到平面的距离相等，说明直线和平面平行。

如果直线和平面相交，如何计算交点？

如图11.24所示，在 x、y、z 轴的任何一个方向上都有 $\frac{p-p_1}{p-p_2}=\frac{y}{x}$，推导得 $p=\frac{p_1*x-p_2*y}{x-y}$。

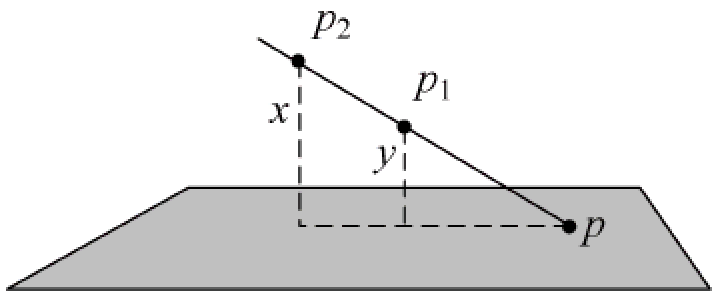


图11.24　直线和平面的交点

7. 四面体的有向体积

四面体是最简单的立体结构。四面体的体积等于底面三角形面积乘以高的1/3，利用叉积和点积很容易计算，代码如下：

```
//四面体有向体积×6
double volume4(Point3 a,Point3 b,Point3 c,Point3 d){
    return Dot(Cross(b - a,c - a),d - a); }
```

【习题】

hdu 1140/4617。

hdu 5733，四面体。

11.3.4 最小球覆盖

最小球覆盖问题：给定 n 个点的三维坐标，求一个半径最小的球，把 n 个点全部包围进来。

和最小圆覆盖一样，最小球覆盖问题也有两种解法，即几何算法和模拟退火算法。

1. 模拟退火算法

如果数据规模较小，可以用模拟退火算法求最小球覆盖。其代码和最小圆覆盖的程序几乎一样，只需加上对坐标 z 的处理即可。

2. 几何算法

和最小圆覆盖增量法的思路类似，最小球覆盖也可以由一些点来确定。一个三维空间中的球，需要 1～4 个点来确定。可以从一个点开始，每次加入一个新的点，更新最小球，直到扩展到全部 n 个点。设前 i 个点的最小覆盖球是 C_i，简单说明如下：

(1) 1 个点。C_1 的球心就是 p_1，半径为 0。

(2) 2 个点。新的 C_2 的球心是线段 p_1p_2 的中心，半径为两点距离的一半。

(3) 3 个点。3 个点构成的平面一定是球的大圆所在的平面，所以球心是三角形的外心，半径就是球心到某个点的距离。

(4) 4 个点。若 4 个点共面则转化到(3)，考虑某 3 个点的情况，若 4 点不共面，四面体可以唯一确定一个外接球。

(5) 对于 5 个及以上点，其最小球必为其中某 4 个点的外接球。

最小覆盖球的代码比较复杂。读者可以通过下面的例题来了解最小覆盖球问题的几何算法。

poj 2069 "Super Star"
输入 n 个点的三维坐标，$4 \leqslant n \leqslant 30$，求最小球覆盖，输出球的半径。

11.3.5 三维凸包

三维凸包问题：给定三维空间的一些点，找到包含这些点的最小凸多面体。三维凸包问题是二维凸包问题的扩展，它是一个比较难的问题。

如果用暴力法求三维凸包，可以枚举任意 3 个点组成的三角形，判断其他点是否都在三角形构成的平面的一侧，如果是，则这个三角形是凸包的一个面。

三维凸包的常用算法是增量法。该算法的思想和最小圆覆盖的增量法有些类似，即把点一个个加入到凸包中。首先找到 4 个不共线、不共面的点，一起构成一个四面体，这是初始凸包，然后依次检查其他点，看这个点是否能在原凸包的基础上构成新的凸包。例如，当检查到点 p_i 时有两种情况：

(1) 如果 p_i 在当前的凸包内，忽略它。

(2) 如果 p_i 不在凸包内，说明用 p_i 可以更新凸包。具体做法是从 p_i 点向凸包看去，将能看到的面全部删除，并把 p_i 和留下的轮廓组合成新的面，填补被删除的面。

三维凸包题目的相关问题有凸包有几个表面、凸包的表面积、凸包的重心等。

复杂度。如果给定的点是随机排列的，算法的期望时间是 $O(n\log_2 n)$[①]的。

下面用一个例题给出三维凸包的模板代码。

hdu 3662 "3D Convex Hull"

输入 n 个点的三维坐标，求三维凸包有几个面。

代码如下[②]：

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 1050;
const double eps = 1e-8;
struct Point3{                                   //三维：点
    double x,y,z;
    Point3(){}
    Point3(double x,double y,double z):x(x),y(y),z(z){}
    Point3 operator + (Point3 B){return Point3(x+B.x,y+B.y,z+B.z);}
    Point3 operator - (Point3 B){return Point3(x-B.x,y-B.y,z-B.z);}
    Point3 operator * (double k){return Point3(x*k,y*k,z*k);}
    Point3 operator / (double k){return Point3(x/k,y/k,z/k);}
};
typedef Point3 Vector3;
double Dot(Vector3 A,Vector3 B){return A.x*B.x+A.y*B.y+A.z*B.z;}
Point3 Cross(Vector3 A,Vector3 B){
    return Point3(A.y*B.z-A.z*B.y,A.z*B.x-A.x*B.z,A.x*B.y-A.y*B.x);}
double Len(Vector3 A){return sqrt(Dot(A,A));}   //向量的长度
double Area2(Point3 A,Point3 B,Point3 C){return Len(Cross(B-A, C-A));}
//四面体有向体积×6
double volume4(Point3 A,Point3 B,Point3 C,Point3 D){
    return Dot(Cross(B-A,C-A),D-A);}
struct CH3D{
    struct face{
        int a,b,c;                               //凸包一个面上的 3 个点的编号
        bool ok;                                 //该面是否在最终凸包上
    };
    int n;                                       //初始顶点数
    Point3 P[MAXN];                              //初始顶点
    int num;                                     //凸包表面的三角形数
    face F[8*MAXN];                              //凸包表面的三角形
    int g[MAXN][MAXN];                           //点 i 到点 j 属于哪个面
    //点在面的同向
    double dblcmp(Point3 &p,face &f){
        Point3 m = P[f.b] - P[f.a];
        Point3 n = P[f.c] - P[f.a];
```

① 证明见《计算几何算法与应用(第 3 版)》，Mark de Berg 等著，邓俊辉译，清华大学出版社，257 页。

② 此代码中的 CH3D()是流传很广的经典模板，如果 CH3D()的原作者看到这里，请联系本书作者。

```
    Point3 t = p - P[f.a];
    return Dot(Cross(m,n),t);
}
void deal(int p,int a,int b){
    int f = g[a][b];                          //搜索与该边相邻的另一个平面
    face add;
    if(F[f].ok){
        if(dblcmp(P[p],F[f])> eps)
            //如果从 p 点能看到该面 f,则继续深度探索 f 的 3 条边,以更新新的凸面
            dfs(p,f);
        else{
            //如果从 p 点看不到 f 面,则 p 点和 a、b 点组成一个三角形
            add.a = b;
            add.b = a;
            add.c = p;
            add.ok = true;
            g[p][b] = g[a][p] = g[b][a] = num;
            F[num++] = add;
        }
    }
}
void dfs(int p,int now){                      //维护凸包,如果点 p 在凸包外则更新凸包
    F[now].ok = 0;
    deal(p,F[now].b,F[now].a);
    deal(p,F[now].c,F[now].b);
    deal(p,F[now].a,F[now].c);
}
bool same(int s,int t){                       //判断两个面是否为同一面
    Point3 &a = P[F[s].a];
    Point3 &b = P[F[s].b];
    Point3 &c = P[F[s].c];
    return fabs(volume4(a,b,c,P[F[t].a]))< eps &&
          fabs(volume4(a,b,c,P[F[t].b]))< eps &&
          fabs(volume4(a,b,c,P[F[t].c]))< eps;
}
//构建三维凸包
void create(){
    int i,j,tmp;
    face add;
    num = 0;
    if(n < 4)return;
    //前 4 个点不共面
    bool flag = true;
    for(i = 1;i < n;i++){                     //使前两个点不共点
        if(Len(P[0] - P[i])> eps){
            swap(P[1],P[i]);
            flag = false;
            break;
        }
    }
    if(flag)return;
```

```
    flag = true;
    //使前 3 个点不共线
    for(i = 2;i < n;i++){
        if(Len(Cross(P[0] - P[1],P[1] - P[i]))> eps){
            swap(P[2],P[i]);
            flag = false;
            break;
        }
    }
    if(flag)return;
    flag = true;
    //使前 4 个点不共面
    for(int i = 3;i < n;i++){
        if(fabs(Dot(Cross(P[0] - P[1], P[1] - P[2]),P[0] - P[i]))> eps){
            swap(P[3],P[i]);
            flag = false;
            break;
        }
    }
    if(flag)return;
    for(i = 0;i < 4;i++){              //构建初始四面体(4 个点为 p[0]、p[1]、p[2]、p[3])
        add.a = (i + 1) % 4;
        add.b = (i + 2) % 4;
        add.c = (i + 3) % 4;
        add.ok = true;
        if(dblcmp(P[i],add)> 0)swap(add.b,add.c);
            //保证逆时针,即法向量朝外,这样新点才可看到
        g[add.a][add.b] = g[add.b][add.c] = g[add.c][add.a] = num;
            //逆向的有向边保存
        F[num++] = add;
    }
    for(i = 4;i < n;i++){              //构建更新凸包
        for(j = 0;j < num;j++){
          //判断点是否在当前三维凸包内,i 表示当前点,j 表示当前面
            if(F[j].ok&&dblcmp(P[i],F[j])> eps){
                    //对当前凸包面进行判断,看点能否看到这个面
                dfs(i,j);              //点能看到当前面,更新凸包的面
                break;
            }
        }
    }
    tmp = num;
    for(i = num = 0;i < tmp;i++)
      if(F[i].ok)
        F[num++] = F[i];
}
//凸包的表面积
double area(){
    double res = 0;
    for(int i = 0;i < num;i++)
      res += Area2(P[F[i].a],P[F[i].b],P[F[i].c]);
```

```
            return res/2.0;
        }
        //体积
        double volume(){
            double res = 0;
            Point3 tmp(0,0,0);
            for(int i = 0;i < num;i++)
               res += volume4(tmp,P[F[i].a],P[F[i].b],P[F[i].c]);
            return fabs(res/6.0);
        }
        //表面三角形个数
        int triangle(){
            return num;
        }
        //表面多边形个数
        int polygon(){
            int i,j,res,flag;
            for(i = res = 0;i < num;i++){
                flag = 1;
                for(j = 0;j < i;j++)
                  if(same(i,j)){
                      flag = 0;
                      break;
                  }
                res += flag;
            }
            return res;
        }
};
CH3D hull;
int main(){
        while(scanf("%d",&hull.n) == 1){
            for(int i = 0;i < hull.n;i++)
                scanf("%lf%lf%lf",&hull.P[i].x,&hull.P[i].y,&hull.P[i].z);
            hull.create();
            printf("%d\n",hull.polygon());
        }
        return 0;
}
```

【习题】

hdu 4273,三维凸包重心。

hdu 3662。

11.4 几何模板

视频讲解

下面给出了本章的模板代码,以便于用户编程时参考。

```
const double pi = acos(-1.0);           //高精度圆周率
const double eps = 1e-8;                //偏差值
const int maxp = 1010;                  //点的数量
int sgn(double x){                      //判断 x 是否等于 0
    if(fabs(x) < eps)  return 0;
    else return x<0?-1:1;
}
int Dcmp(double x, double y){           //比较两个浮点数: 0 为相等; -1 为小于; 1 为大于
    if(fabs(x - y) < eps) return 0;
    else return x<y ?-1:1;
}
//---------------平面几何: 点和线---------------------------
struct Point{                           //定义点及其基本运算
    double x,y;
    Point(){}
    Point(double x,double y):x(x),y(y){}
    Point operator + (Point B){return Point(x+B.x,y+B.y);}
    Point operator - (Point B){return Point(x-B.x,y-B.y);}
    Point operator * (double k){return Point(x*k,y*k);}     //长度增大 k 倍
    Point operator / (double k){return Point(x/k,y/k);}     //长度缩小 k 倍
    bool operator == (Point B){return sgn(x-B.x) == 0 && sgn(y-B.y) == 0;}
    bool operator < (Point B){                               //比较两个点,用于凸包计算
        return sgn(x-B.x)<0 || (sgn(x-B.x) == 0 && sgn(y-B.y)<0);}
};
typedef Point Vector;                                        //定义向量
double Dot(Vector A,Vector B){return A.x*B.x + A.y*B.y;}     //点积
double Len(Vector A){return sqrt(Dot(A,A));}                 //向量的长度
double Len2(Vector A){return Dot(A,A);}                      //向量长度的平方
//A 与 B 的夹角
double Angle(Vector A,Vector B){return acos(Dot(A,B)/Len(A)/Len(B));}
double Cross(Vector A,Vector B){return A.x*B.y - A.y*B.x;}   //叉积
//三角形 ABC 面积的两倍
double Area2(Point A, Point B, Point C){return Cross(B-A, C-A);}
//两点的距离,用两种方式实现
double Distance(Point A, Point B){return hypot(A.x-B.x,A.y-B.y);}
double Dist(Point A,Point B){
    return sqrt((A.x-B.x)*(A.x-B.x) + (A.y-B.y)*(A.y-B.y));}
//向量 A 的单位法向量
Vector Normal(Vector A){return Vector(-A.y/Len(A), A.x/Len(A));}
//向量是否平行或重合
bool Parallel(Vector A, Vector B){return sgn(Cross(A,B)) == 0;}
Vector Rotate(Vector A, double rad){                         //向量 A 逆时针旋转 rad 度
    return Vector(A.x*cos(rad)-A.y*sin(rad), A.x*sin(rad)+A.y*cos(rad));
}
struct Line{
    Point p1,p2;                                             //线上的两个点
    Line(){}
    Line(Point p1,Point p2):p1(p1),p2(p2){}
    //根据一个点和倾斜角 angle 确定直线,0<=angle<pi
    Line(Point p,double angle){
```

```
        p1 = p;
        if(sgn(angle - pi/2) == 0){p2 = (p1 + Point(0,1));}
        else{p2 = (p1 + Point(1,tan(angle)));}
    }
    //ax + by + c = 0
    Line(double a,double b,double c){
        if(sgn(a) == 0){
            p1 = Point(0, - c/b);
            p2 = Point(1, - c/b);
        }
        else if(sgn(b) == 0){
            p1 = Point( - c/a,0);
            p2 = Point( - c/a,1);
        }
        else{
            p1 = Point(0, - c/b);
            p2 = Point(1,( - c - a)/b);
        }
    }
};
typedef Line Segment;                       //定义线段,两端点是 Point p1,p2
//返回直线倾斜角,0≤angle < pi
double Line_angle(Line v){
    double k = atan2(v.p2.y - v.p1.y, v.p2.x - v.p1.x);
    if(sgn(k) < 0)k += pi;
    if(sgn(k - pi) == 0)k -= pi;
        return k;
}
//点和直线的关系:1 为在左侧;2 为点在右侧;0 为点在直线上
int Point_line_relation(Point p,Line v){
    int c = sgn(Cross(p - v.p1,v.p2 - v.p1));
    if(c < 0)return 1;                      //1: p 在 v 的左边
    if(c > 0)return 2;                      //2: p 在 v 的右边
    return 0;                               //0: p 在 v 上
}
//点和线段的关系: 0 为点 p 不在线段 v 上; 1 为点 p 在线段 v 上
bool Point_on_seg(Point p, Line v){
    return sgn(Cross(p - v.p1, v.p2 - v.p1)) == 0 &&
        sgn(Dot(p - v.p1,p - v.p2)) <= 0;
}
//两直线的关系:0 为平行,1 为重合,2 为相交
int Line_relation(Line v1, Line v2){
    if(sgn(Cross(v1.p2 - v1.p1,v2.p2 - v2.p1)) == 0){
        if(Point_line_relation(v1.p1,v2) == 0)  return 1;   //1: 重合
        else return 0;                                      //0: 平行
    }
    return 2;                                               //2: 相交
}
//点到直线的距离
double Dis_point_line(Point p, Line v){
    return fabs(Cross(p - v.p1,v.p2 - v.p1))/Distance(v.p1,v.p2);
```

```
}
//点在直线上的投影
Point Point_line_proj(Point p, Line v){
    double k = Dot(v.p2 - v.p1,p - v.p1)/Len2(v.p2 - v.p1);
    return v.p1 + (v.p2 - v.p1) * k;
}
//点 p 对直线 v 的对称点
Point Point_line_symmetry(Point p, Line v){
    Point q = Point_line_proj(p,v);
    return Point(2 * q.x - p.x,2 * q.y - p.y);
}
//点到线段的距离
double Dis_point_seg(Point p, Segment v){
    if(sgn(Dot(p - v.p1,v.p2 - v.p1))< 0 || sgn(Dot(p - v.p2,v.p1 - v.p2))< 0)
                                                    //点的投影不在线段上
        return min(Distance(p,v.p1),Distance(p,v.p2));
    return Dis_point_line(p,v);                     //点的投影在线段上
}
//求两直线 ab 和 cd 的交点,在调用前要保证两直线不平行或重合
Point Cross_point(Point a,Point b,Point c,Point d){ //Line1:ab,Line2:cd
    double s1 = Cross(b - a,c - a);
    double s2 = Cross(b - a,d - a);                 //叉积有正负
    return Point(c.x * s2 - d.x * s1,c.y * s2 - d.y * s1)/(s2 - s1);
}
//两线段是否相交: 1 为相交,0 为不相交
bool Cross_segment(Point a,Point b,Point c,Point d){ //Line1:ab,Line2:cd
    double c1 = Cross(b - a,c - a),c2 = Cross(b - a,d - a);
    double d1 = Cross(d - c,a - c),d2 = Cross(d - c,b - c);
    return sgn(c1) * sgn(c2)< 0 && sgn(d1) * sgn(d2)< 0;
                                                    //注意交点是端点的情况不算在内
}
//---------------平面几何: 多边形----------------
struct Polygon{
    int n;                                          //多边形的顶点数
    Point p[maxp];                                  //多边形的点
    Line v[maxp];                                   //多边形的边
};
//判断点和任意多边形的关系: 3 为点上; 2 为边上; 1 为内部; 0 为外部
int Point_in_polygon(Point pt,Point * p,int n){     //点 pt,多边形 Point * p
    for(int i = 0;i < n;i++){                       //点在多边形的顶点上
        if(p[i] == pt)return 3;
    }
    for(int i = 0;i < n;i++){                       //点在多边形的边上
        Line v = Line(p[i],p[(i + 1) % n]);
        if(Point_on_seg(pt,v)) return 2;
    }
    int num = 0;
    for(int i = 0;i < n;i++){
        int j = (i + 1) % n;
        int c = sgn(Cross(pt - p[j],p[i] - p[j]));
        int u = sgn(p[i].y - pt.y);
```

```
            int v = sgn(p[j].y - pt.y);
            if(c > 0 && u < 0 && v >= 0) num++;
            if(c < 0 && u >= 0 && v < 0) num--;
        }
        return num != 0;                    //1 为内部; 0 为外部
}
//多边形面积
double Polygon_area(Point *p, int n){  //从原点开始划分三角形
    double area = 0;
    for(int i = 0;i < n;i++)
        area += Cross(p[i],p[(i+1)%n]);
    return area/2;                      //面积有正负,不能简单地取绝对值
}
//求多边形的重心
Point Polygon_center(Point *p, int n){
    Point ans(0,0);
    if(Polygon_area(p,n) == 0) return ans;
    for(int i = 0;i < n;i++)
        ans = ans + (p[i]+p[(i+1)%n]) * Cross(p[i],p[(i+1)%n]);
    return ans/Polygon_area(p,n)/6.;
}
//Convex_hull()求凸包.凸包顶点放在 ch 中,返回值是凸包的顶点数
int Convex_hull(Point *p,int n,Point *ch){
    sort(p,p+n);                        //对点排序: 按 x 从小到大排序,如果 x 相同,按 y 排序
    n=unique(p,p+n)-p;                  //去除重复点
    int v=0;
    //求下凸包.如果 p[i]是右拐弯的,这个点不在凸包上,往回退
    for(int i=0;i<n;i++){
        while(v>1 && sgn(Cross(ch[v-1]-ch[v-2],p[i]-ch[v-2]))<=0)
            v--;
        ch[v++]=p[i];
    }
    int j=v;
    //求上凸包
    for(int i=n-2;i>=0;i--){
        while(v>j && sgn(Cross(ch[v-1]-ch[v-2],p[i]-ch[v-2]))<=0)
            v--;
        ch[v++]=p[i];
    }
    if(n>1) v--;
    return v;                           //返回值 v 是凸包的顶点数
}

//---------------平面几何: 圆----------------
struct Circle{
    Point c;                            //圆心
    double r;                           //半径
    Circle(){}
    Circle(Point c,double r):c(c),r(r){}
    Circle(double x,double y,double _r){c=Point(x,y);r = _r;}
};
```

```
//点和圆的关系：0 为点在圆内，1 为点在圆上，2 为点在圆外
int Point_circle_relation(Point p, Circle C){
    double dst = Distance(p,C.c);
    if(sgn(dst - C.r) < 0) return 0;            //点在圆内
    if(sgn(dst - C.r) == 0) return 1;           //圆上
    return 2;                                   //圆外
}
//直线和圆的关系：0 为直线在圆内，1 为直线和圆相切，2 为直线在圆外
int Line_circle_relation(Line v,Circle C){
    double dst = Dis_point_line(C.c,v);
    if(sgn(dst - C.r) < 0) return 0;            //直线在圆内
    if(sgn(dst - C.r) == 0) return 1;           //直线和圆相切
    return 2;                                   //直线在圆外
}
//线段和圆的关系：0 为线段在圆内，1 为线段和圆相切，2 为线段在圆外
int Seg_circle_relation(Segment v,Circle C){
    double dst = Dis_point_seg(C.c,v);
    if(sgn(dst - C.r) < 0) return 0;            //线段在圆内
    if(sgn(dst - C.r) == 0) return 1;           //线段和圆相切
    return 2;                                   //线段在圆外
}
//直线和圆的交点.pa、pb 是交点.返回值是交点的个数
int Line_cross_circle(Line v,Circle C,Point &pa,Point &pb){
    if(Line_circle_relation(v, C) == 2) return 0;  //无交点
    Point q = Point_line_proj(C.c,v);           //圆心在直线上的投影点
    double d = Dis_point_line(C.c,v);           //圆心到直线的距离
    double k = sqrt(C.r * C.r - d * d);
    if(sgn(k) == 0){                            //一个交点,直线和圆相切
        pa = q;
        pb = q;
        return 1;
    }
    Point n = (v.p2 - v.p1)/ Len(v.p2 - v.p1);  //单位向量
    pa = q + n * k;
    pb = q - n * k;
    return 2;                                   //两个交点
}

//------------------- 三维几何 ----------------
//三维：点
struct Point3{
    double x,y,z;
    Point3(){}
    Point3(double x,double y,double z):x(x),y(y),z(z){}
    Point3 operator + (Point3 B){return Point3(x + B.x,y + B.y,z + B.z);}
    Point3 operator - (Point3 B){return Point3(x - B.x,y - B.y,z - B.z);}
    Point3 operator * (double k){return Point3(x * k,y * k,z * k);}
    Point3 operator / (double k){return Point3(x/k,y/k,z/k);}
    bool operator == (Point3 B){
        return sgn(x - B.x) == 0 && sgn(y - B.y) == 0 && sgn(z - B.z) == 0;}
};
```

```
typedef Point3 Vector3;
//点积.和二维点积函数同名.C++允许函数同名
double Dot(Vector3 A,Vector3 B){return A.x * B.x + A.y * B.y + A.z * B.z;}
//叉积
Vector3 Cross(Vector3 A,Vector3 B){
    return Point3(A.y * B.z - A.z * B.y,A.z * B.x - A.x * B.z,A.x * B.y - A.y * B.x);}
double Len(Vector3 A){return sqrt(Dot(A,A));}        //向量的长度
double Len2(Vector3 A){return Dot(A,A);}              //向量长度的平方
double Distance(Point3 A, Point3 B){                  //A、B 的距离
    return sqrt((A.x - B.x) * (A.x - B.x) +
                (A.y - B.y) * (A.y - B.y) + (A.z - B.z) * (A.z - B.z));
}
//A 与 B 的夹角
double Angle(Vector3 A,Vector3 B){return acos(Dot(A,B)/Len(A)/Len(B));}
//三维：线
struct Line3{
    Point3 p1,p2;
    Line3(){}
    Line3(Point3 p1,Point3 p2):p1(p1),p2(p2){}
};
typedef Line3 Segment3;                               //定义线段,两端点是 Point p1,p2
//三维：三角形面积的两倍
double Area2(Point3 A,Point3 B,Point3 C){return Len(Cross(B - A, C - A));}
//三维：点到直线的距离
double Dis_point_line(Point3 p, Line3 v){
    return Len(Cross(v.p2 - v.p1,p - v.p1))/Distance(v.p1,v.p2);
}
//三维：点在直线上
bool Point_line_relation(Point3 p,Line3 v){
    return sgn(Len(Cross(v.p1 - p,v.p2 - p)))  ==  0
                    && sgn(Dot(v.p1 - p,v.p2 - p)) ==  0;
}
//三维：点到线段的距离
double Dis_point_seg(Point3 p, Segment3 v){
    if(sgn(Dot(p - v.p1,v.p2 - v.p1)) < 0 || sgn(Dot(p - v.p2,v.p1 - v.p2)) < 0)
        return min(Distance(p,v.p1),Distance(p,v.p2));
    return Dis_point_line(p,v);
}
//三维：点 p 在直线上的投影
Point3 Point_line_proj(Point3 p, Line3 v){
    double k = Dot(v.p2 - v.p1,p - v.p1)/Len2(v.p2 - v.p1);
    return v.p1 + (v.p2 - v.p1) * k;
}
//三维：平面
struct Plane{
    Point3 p1,p2,p3;                                  //平面上的 3 个点
    Plane(){}
    Plane(Point3 p1,Point3 p2,Point3 p3):p1(p1),p2(p2),p3(p3){}
};
//平面法向量
Point3 Pvec(Point3 A,Point3 B,Point3 C){ return Cross(B - A,C - A);}
```

```
Point3 Pvec(Plane f){return Cross(f.p2 - f.p1,f.p3 - f.p1);}
//四点共平面
bool Point_on_plane(Point3 A,Point3 B,Point3 C,Point3 D){
    return sgn(Dot(Pvec(A,B,C),D - A)) == 0;
}
//两平面平行
int Parallel(Plane f1, Plane f2){
    return Len(Cross(Pvec(f1),Pvec(f2))) < eps;
}
//两平面垂直
int Vertical (Plane f1, Plane f2){
    return sgn(Dot(Pvec(f1),Pvec(f2))) == 0;
}
//直线与平面的交点 p,返回值是交点的个数
int Line_cross_plane(Line3 u,Plane f,Point3 &p){
    Point3 v = Pvec(f);                          //平面的法向量
    double x = Dot(v, u.p2 - f.p1);
    double y = Dot(v, u.p1 - f.p1);
    double d = x - y;
    if(sgn(x) == 0 && sgn(y) == 0) return -1;    //-1: v 在 f 上
    if(sgn(d) == 0) return 0;                    //0: v 与 f 平行
    p = ((u.p1 * x) - (u.p2 * y))/d;             //1: v 与 f 相交
    return 1;
}
//四面体有向体积×6
double volume4(Point3 A,Point3 B,Point3 C,Point3 D){
    return Dot(Cross(B - A,C - A),D - A);}
```

11.5　小　结

几何题是竞赛初学者的难关，很多竞赛队甚至没有队员深入研究几何题，以至于在赛场上看到几何题直接放弃。

几何题往往逻辑烦琐，需要细致地编程，很考验编码能力。本章提到的内容，竞赛队的所有队员都应精通，并且指定其中一人深入研究，做到能轻松解决中等难度以上的题目。

第 12 章　ICPC 区域赛真题

前面各章节讲解了竞赛所需要的知识点。在参赛之前，队员有必要了解 ICPC 区域赛各赛区的总体情况，并通过解读 ICPC 区域赛真题做到心中有数，确定一个冲刺的目标。

历年的世界总决赛（World Finals）题目、区域赛题目都可以在官网 icpc. baylor. edu 浏览和提交①。其中，中国大陆往年的 ICPC 区域赛题目，acm. hdu. edu. cn 也有收录。

在一次区域赛中，为了区分参赛队员的能力，出题者需要考虑多方面的因素才能出一套合适的题目。大体上应该能考查竞赛队员的 5 种能力，包括编码、计算思维、逻辑推理、算法知识、团队合作。难度上的区分如表 12.1 所示。

表 12.1　难度上的区分

奖牌	编码	计算思维	逻辑推理	算法知识	团队合作	总分
铜牌	**	**	**	**	*	9
银牌	****	***	***	***	***	16
金牌	*****	*****	*****	*****	*****	25

从能力考查上看，铜牌 1 星或 2 星；银牌 3 星或 4 星；金牌 5 星。再考虑到铜牌、银牌、金牌的获奖比例按 3：2：1 逐渐缩小，可以得出结论：从铜牌提升到银牌比较容易，从银牌提升到金牌很难。

奖牌和参赛队 3 个队员的能力直接相关。如果有一人编码快，做题熟练，靠他一人就可以得到铜牌；如果 3 人都能进行大量训练，编码得心应手，算法知识大量掌握，加上一定的团队合作，可以得到银牌；在银牌的基础上，如果 3 人在平时训练中喜欢深入思考，不怕复杂的编码，把多种算法和数据结构融会贯通，团队合作紧密，再经历长期的练习，可以冲击金牌。

金牌、银牌队员是未来的 IT 精英，铜牌队员则需要继续努力。

12.1　ICPC 亚洲区域赛（中国大陆）情况

每年中国大陆赛区有 4～6 个，每个赛区的参赛队伍有 250 个左右，参赛学校 100 多个。奖牌设置比例是金牌 10%、银牌 20%、铜牌 30%，共 60% 的参赛队得奖。例如，2015 年、2017 年亚洲区域赛中国大陆赛区相关信息见表 12.2 和表 12.3。

① ICPC live archive，网上简称 LA（https://icpcarchive. ecs. baylor. edu/）。

表 12.2　2015 年亚洲区域赛中国大陆赛区各站比赛情况统计

2015 年	长春	沈阳	合肥	北京	上海	EC-Final
题目数量	13	13	10	11	12	13
金牌题数	5～8	5～8	4～7	7～8	6～9	≥7
银牌题数	4～5	4～5	2～4	5～6	5～6	5～7
铜牌题数	4	3	1	4	4～5	3～5
参赛队数	220	172	97	200	199	288

表 12.3　2017 年亚洲区域赛中国大陆赛区各站比赛情况统计

2017 年	沈阳	西安	青岛	北京	南宁	乌鲁木齐	EC-Final
题目数量	13	11	11	10	11	10	13
金牌题数	6～10	6～10	3～6	5～9	8～11	7～10	9～11
银牌题数	5～6	4～6	3	3～5	6～7	5～7	6～9
铜牌题数	4～5	3～4	2～3	2～3	4～6	3～5	4～6
参赛队数	180	350	360	190	228	94	300

12.2　ICPC 区域赛题目解析

视频讲解

2015 年 11 月 22 日，亚洲区域赛上海站于华东理工大学举行，共有 120 所大学、199 队参加，来自中国大陆、中国香港（4 校 7 队），以及朝鲜（1 校 1 队）、蒙古（1 校 3 队）等国家和地区，是一次典型的、有影响力的亚洲区域赛。

题目由电子科技大学退役金牌队员张鑫航、何云鹏拟定。题目难度分布合理，有很好的区分度①。

本次比赛共 12 道题目，题目在 ICPC 官网有存档②，或者在 hdu 上提交，题号是 5572～5584。读者在看下面的内容之前，为了更好地理解题目，提高自己的思考能力，最好先自己尝试做题。

铜牌：4 道或 5 道题，F、K、L、A、B 题。

银牌：5 道或 6 道题，加上 D 题。

金牌：6 道题以上，加上 E、G、I 题。

C、H、J 题有部分做出。

参赛队解题时间（AC 时间）如表 12.4 所示。

① 现场赛排名：https://perma.cc/VJ2A-B642。
官方档案：https://icpc.baylor.edu/regionals/finder/shanghai-2015/standings（短网址：t.cn/R397ena）。

② https://icpcarchive.ecs.baylor.edu/index.php?option=com_onlinejudge&Itemid=8&category=691（短网址：t.cn/R39zGy4）。

表 12.4　AC 时间(分钟,中位值)

题　　目	F	K	L	A	B	D	E	G	I
金牌(≥6 道)	7	25	48	101	97	239	<u>229</u>①	<u>269</u>	<u>270</u>
银牌(5 道或 6 道)	8	40	64	140	146	<u>255</u>	<u>184</u>	<u>188</u>	
铜牌(4 道或 5 道)	10	40	98	<u>218</u>	<u>213</u>				

下面按难易程度,由简到难对 9 道题目进行详细的讲解。

12.2.1　F 题 Friendship of Frog(hdu 5578)

Time Limit：2000/1000ms(Java/Others)

Memory Limit：65536/65536KB(Java/Others)②

Problem Description：

N 只青蛙站成一排,它们来自不同的国家。每个国家用一个小写字母表示。相邻青蛙的距离(例如第 1 个和第 2 个青蛙、第 $N-1$ 和第 N 个青蛙等)是 1。如果两只青蛙来自同一个国家,那么它们是朋友。

距离最小的一对朋友是最亲密的。帮忙找出这个距离是多少。

Input：

第 1 行是一个整数 T,表示测试用例的个数。

每一个测试用例只包括一串长度为 N 的字符串,其中第 i 个字符表示第 i 个青蛙的国籍。

Output：

对每个测试用例,需要输出"Case ＃x:y",其中 x 表示第几个用例,从 1 开始计数；y 是结果。如果没有来自同一个国家的青蛙,输出－1。

Limits：

$1 \leqslant T \leqslant 50$；

80％的数据,$1 \leqslant N \leqslant 100$；

100％的数据,$1 \leqslant N \leqslant 1000$；

字符串只包括小写字母。

Sample Input	Sample Output
2 abcecba abc	Case ＃1：2 Case ＃2：－1

【题解】

难度等级：极简单。本题是所谓的“签到题”,即参赛的所有队伍都能 AC 的简单题。

---

① 画线表示只有部分队伍做出来。

② Time Limit 和 Memory Limit 是本题的时间和空间限制,但是现场赛所发的题目一般不会给出,需要参赛队自己判断是否超时和超内存。这和在线判题的 OJ 不同,为方便学习,OJ 一般会给出这两个参数。Time Limit 和编程语言也有关系,Java 程序比 C、C++程序慢,所以 Java 的 Time Limit 更大一些,一般是 C、C++的 3 倍以上。

从读题到提交代码，参赛队 AC 的最短时间(First Blood，FB)是 3 分钟。铜牌队伍在 10 分钟左右 AC。

能力考核：编码能力。

本题逻辑简单，很容易理解，不涉及复杂的算法，代码也很短，参赛队员只要了解竞赛的入门知识就能做出来。

(1) 时间复杂度。在读题时，首先注意到数据长度 N 很小，$1 \leqslant N \leqslant 1000$，因此可以采用时间复杂度为 $O(N^3)$ 的算法。

(2) 逻辑和算法。由于题目很简单，首先想到暴力的方法。思路是从头开始检查第 i 个字符($0 \leqslant i \leqslant N$)，逐个比对它后面的 $N-i$ 个字符，寻找相同的。每次比对时把最短距离记录下来，直到全部结束。其复杂度是 $O(N^2)$，所以这个方法是可行的。

(3) 扩展。

虽然该题是很简单的题目，但是如果数据很大，例如 $N \leqslant 10^5$，那么上述的方法就会超时，此时需要更好的思路。

用暴力法结合“剪枝”的技巧可以处理。下面示例程序的时间复杂度是 $O(N)$。

```
#include <bits/stdc++.h>
using namespace std;
const int N = 100010;
char s[N];
void solve() {
    scanf("%s", s);
    int n = strlen(s);
    int ans = -1;
    for(int i = 0; i < n; ++i) {          //从头到尾检查字符串,i 是当前位置
        for(int j = 1; j <= 26 && i - j >= 0; ++j) {
                //剪枝技巧: 由于小写字符一共有 26 个,所以字符串中两个相
                //同字符的最小距离不会超过 26,只需要检查 i 前面的 26 个字符
            if(s[i] == s[i - j]) {
                //j 从 1 开始递增,即从距离 i 最近的字符开始检查,
                //如果有相同字符,就 break; 其他未检查的距离更大,不用继续检查
                if(ans == -1 || j < ans) {
                    ans = j;
                }
                break;
            }
        }
    }
    printf("%d\n", ans);
}
int main() {
    int t;
    scanf("%d", &t);
    for(int i = 1; i <= t; ++i) {
        printf("Case #%d: ", i);
        solve();
    }
    return 0;
}
```

12.2.2 K 题 Kingdom of Black and White(hdu 5583)

Problem Description：

黑白国有两种青蛙：黑青蛙和白青蛙。N 只青蛙站成一行，有些是黑的，有些是白的。计算青蛙们的合力，计算规则如下：把青蛙们分成最小的部分，每部分是连续的，只包含一种颜色的青蛙；合力是每部分长度的平方和。

现在来了一个罪恶的老巫婆，她告诉青蛙们，她要改变青蛙的颜色，最多改变一只。这样青蛙们的合力就变了。

青蛙们想知道，巫婆完成她的工作后，可能的最大合力是多少。

Input：

第 1 行是一个整数 T，表示测试用例的个数。

每个测试用例只包含一个字符串，长度为 N，只包含字符'0'(表示一只黑青蛙)和'1'(表示一只白青蛙)。

Output：

对每个测试用例，需要输出"Case #x:y"，其中 x 表示第几个用例，从 1 开始计数；y 是结果。

Limits：

$1 \leqslant T \leqslant 50$；

60%的数据，$1 \leqslant N \leqslant 1000$；

100%的数据，$1 \leqslant N \leqslant 10^5$；

字符串值包含 0 和 1。

Sample Input	Sample Output
2 000011 0101	Case #1：26 Case #2：10

【题解】

难度等级：简单题，用暴力法求解。FB 时间是 10 分钟。铜牌队伍在 40 分钟左右 AC。

能力考核：计算复杂度、编码能力。

本题逻辑虽然简单，但是也需要灵活处理；不涉及复杂的算法，但是需要了解计算的复杂度并避免落入陷阱；代码比较短，但是有一定的技巧。本题可以考查基本的计算思维和较好的编码能力。

(1) 逻辑。根据 Sample Input 和 Sample Output 理解题意，当输入 000011 时，青蛙的合力是 $4^2+2^2=20$。改变一只青蛙的颜色，例如改成 000001，合力变为 $5^2+1^2=26$。判断最大的合力是 26，并输出。

可能的最大合力，出现在有 $N=10^5$ 只青蛙的情况下，且所有青蛙是一种颜色，此时合力是 $N^2=10^{10}<2^{64}$，可以用 64 位的 long long 类型表示。

(2) 计算复杂度。在解题时，首先应该注意数据的规模，即 $1 \leqslant N \leqslant 10^5$，这说明不能使用时间复杂度大于 $O(N^2)$的算法。

算法竞赛的初学者，如果没有注意到这一限制，就会落入陷阱，用以下简单的暴力方法，结果是 TLE：每改变一只青蛙的颜色，就重新计算合力，计算量是 $O(N)$；从头到尾一共可以改变 N 次；总复杂度是 $O(N^2)$。

TLE 的错误代码

```
#include <bits/stdc++.h>
using namespace std;
const int N = 100010;
char s[N];
int n;
long long get_ans() {                    //计算合力，时间复杂度是 O(N)
    long long ans = 0;
    int cur = -1, len = 0;
    for(int i = 1; i <= n; ++ i) {
        if(s[i] - '0' == cur)
            ++ len;
        else {
            ans += 1LL * len * len;
            len = 1;
            cur = s[i] - '0';
        }
    }
    ans += 1LL * len * len;
    return ans;
}
void solve() {                           //总的时间复杂度是 O(N²)，结果 TLE
    scanf("%s", s + 1);
    n = strlen(s + 1);
    long long ans = get_ans();
    for(int i = 1; i <= n; ++ i) {       //逐个改变青蛙的颜色，可以改变 N 次
        s[i] = '1' + '0' - s[i];         //改变当前青蛙的颜色
        ans = max(ans, get_ans());       //计算合力并得到最大值
        s[i] = '1' + '0' - s[i];         //还原青蛙的颜色
    }
    printf("%lld\n", ans);
}
int main() {
    int t;
    scanf("%d", &t);
    for(int i = 1; i <= t; ++ i) {
        printf("Case #%d: ", i);
        solve();
    }
    return 0;
}
```

(3) 优化和解决。在上述程序中，计算复杂度可以优化。每次改变一只青蛙的颜色后，因为只影响了相邻的青蛙序列，所以并不需要全部重新计算，只计算被影响的这部分就行了。这样，每改变一只青蛙的颜色，计算的时间复杂度差不多是 $O(1)$，总复杂度是 $O(N)$。

正确代码

```
#include<bits/stdc++.h>
using namespace std;
const int N = 100010;
#define square(x)  (x)*(x)
void solve() {
    int i, j=1, k=1;
    char s[N];
    int n;
    long long a[N]={0};
    long long maxsum, oldsum=0;
    scanf("%s", s + 1);
    n = strlen(s + 1);
    for(i=2; i<=n; i++){              //把表示青蛙序列的"01"字符串改成数字,例如
         //把"000011001"改成数字 4221,并存放在数组 a[]中,处理起来更加便捷
      if(s[i]==s[i-1])
         j++;
      else {
        a[k]=j;
        k++;
        j=1;
      }
    }
    a[k]=j;
    for(i=1; i<=k; i++){              //计算改变颜色前的合力
      oldsum = oldsum + a[i]*a[i];
    }
    maxsum = oldsum;
    for(i=1; i<=k; i++){              //改变一只青蛙的颜色对合力的影响
                                      //只需要考虑以下两种情况:
      if(a[i]==1)                     //如果长度是1,说明这只青蛙是孤立的,
                                      //改变它的颜色后,可以和左右合并,
                                      //例如"00100"合并成"00000"
        maxsum = max(maxsum, oldsum+square(a[i-1]+a[i]+a[i+1])-square(a[i-1])-
square(a[i])-square(a[i+1]));
      if(a[i]>=2){                    //如果长度大于等于2,可以分两次改变颜色:
           //改变最左边的,与左边的邻居合并,例如"0110"改成"0010";
           //改变最右边的,和右边的邻居合并,例如"0110"改成"0100"
           //如果长度大于等于3,改变中间的,只会减小合力,
           //所以不用考虑,例如"01110"改成"01010",合力变小
        maxsum = max(maxsum, oldsum+square(a[i-1]+1)+square(a[i]-1)-square(a[i-
1])-square(a[i]));                    //给左边
        maxsum = max(maxsum, oldsum+square(a[i+1]+1)+square(a[i]-1)-square(a[i+
1])-square(a[i]));                    //给右边
      }
    }
    printf("%lld\n", maxsum);
}
int main() {
```

```
    int t;
    scanf("%d", &t);
    for(int i = 1; i <= t; ++ i) {
        printf("Case #%d: ", i);
        solve();
    }
    return 0;
}
```

12.2.3 L 题 LCM Walk(hdu 5584)

Problem Description:

一只青蛙刚学会一些数论就迫不及待地想展示给女朋友看。

它坐在一个网格图上,行和列都是无限的。行的计数从底部开始,列也是这样。青蛙最初的位置是坐标(s_x,s_y),旅程开始了。

为了向女朋友炫耀它的数学天才,它使用了一种特别的跳跃方法。如果它在坐标(x,y)上,寻找一个可以被 x 和 y 都整除的最小的 z,然后向上或向右跳 z 步,下一步坐标可能是($x+z$,y)或(x,$y+z$)。

经过有限次跳跃后(可能是 0 步),它停在(e_x,e_y)处。然而它太累了,忘记了它的起始位置。

如果一个个去检查网格的所有坐标,那太笨了!请告诉青蛙一个聪明的办法,到达(e_x,e_y)的可能的起始位置有多少个?

Input:

第 1 行是一个整数 T,表示测试用例的个数。

每个测试用例包含两个整数 e_x、e_y,即目的地坐标。

Output:

对每个测试用例,需要输出"Case #x: y",其中 x 表示第几个用例,从 1 开始计数;y 是可能起点的个数。

Limits:

$1 \leqslant T \leqslant 1000$;

$1 \leqslant e_x, e_y \leqslant 10^9$。

Sample Input	Sample Output
3 6 10 6 8 2 8	Case #1: 1 Case #2: 2 Case #3: 3

【题解】

难度等级:简单题,数学。FB 时间是 18 分钟。铜牌队伍在 98 分钟左右 AC。

能力考核:数论中的最小公倍数和最大公约数问题、逻辑推理。

本题涉及了算法知识,不过比较容易,是简单的数论概念;推导过程需要有清晰、灵活

的推理；需要了解计算的复杂度；代码比较短。本题可以考查基本的算法知识和一定的逻辑推理能力。

(1) 算法复杂度。在解题时，首先应该注意数据的规模，即 $1 \leqslant e_x, e_y \leqslant 10^9$，这说明不能使用时间复杂度大于 $O(N)$ 的算法。

(2) 算法概念和逻辑推理。标题说明这是一个 LCM(最小公倍数)问题。

起点是 (x,y)，终点是 (e_x,e_y)，已知终点，反推起点。

z 是 x,y 的 LCM，假设 $x=pt, y=qt, z=pqt$。p 和 q 互质。

起点 (x,y)，下一步可以走到两个位置 $(x,y+z)$ 或 $(x+z,y)$，下面分别讨论。

① 终点 $(e_x,e_y)=(x,y+z)=(pt,qt+pqt)=(pt,q(1+p)t)$。

由于 p 和 $q(1+p)$ 互质，所以 t 是最大公约数，可得 $t=\text{GCD}(e_x,e_y)$。推导得到起点：

$$x=pt=e_x$$

$$y=qt=e_yt/(e_x+t)$$

这是一个可能的起点。

把起点当成新的终点，继续这个过程，直到结束。

需要注意，p、q、t 都是整数，在程序中需要判断。

② 终点 $(e_x,e_y)=(x+z,y)$。

实际情况和①类似。注意到①中 $y+z>x$，即 $e_x<e_y$，②中 $e_x>e_y$。在编程时，只需要先按大小交换 e_x、e_y 的顺序，就可以合并成一种情况处理了。

```
#include <bits/stdc++.h>
int solve(long long ex, long long ey) {
  int ans = 1;
  long long t;
  while (true) {
     if (ex > ey)
    std::swap(ex, ey);
    t = std::__gcd(ex, ey);
              //p = ex / t;  q = ey/(ex + t); //在计算中 p 和 q 并未用到
    if ((ey % (ex + t)) == 0 ) {               //判断 q 是否为整数; t 和 p 肯定是整数,不用判断
        ey = ey * t/(ex + t);
        ans++;
    }
    else
        break;
  }
  return ans;
}
int main() {
    int T;
    long long ex, ey;
    scanf("%d", &T);
    for (int cas = 1; cas <= T; cas++) {
        scanf("%lld%lld", &ex, &ey);
        printf("Case #%d: %d\n", cas, solve(ex, ey));
    }
    return 0;
}
```

12.2.4 A 题 An Easy Physics Problem(hdu 5572)

Problem Description:

在一个无限光滑的桌面上有一个固定的大圆柱体,还有一个体积忽略不计的小球。

开始时,球静止于 A 点,给它一个初始速度和方向,如果球撞到圆柱体,它会弹回,没有能量损失。

经过一段时间,小球是否会经过 B 点?

Input:

第 1 行是一个整数 T,表示测试用例的个数。

每个测试用例有 3 行。

第 1 行有 3 个整数 Ox、Oy、r。圆柱体的中心是(Ox,Oy),半径为 r。

第 2 行有 4 个整数 Ax、Ay、Vx、Vy。A 的坐标是(Ax,Ay),初始方向矢量是(Vx,Vy)。

第 3 行有两个整数 Bx、By。B 的坐标是(Bx,By)。

Output:

对每个测试用例,需要输出"Case #x: y",其中 x 表示第几个用例,从 1 开始计数;如果球会经过 B 点,y 是"Yes",否则 y 是"No"。

Limits:

$1 \leqslant T \leqslant 100$;

$|\text{Ox}|, |\text{Oy}| \leqslant 1000$;

$1 \leqslant r \leqslant 100$;

$|\text{Ax}|, |\text{Ay}|, |\text{Bx}|, |\text{By}| \leqslant 1000$;

$|\text{Vx}|, |\text{Vy}| \leqslant 1000$;

$\text{Vx} \neq 0$ 或 $\text{Vy} \neq 0$;

A 和 B 都在圆柱体的外面,而且不重合。

Sample Input	Sample Output
2 0 0 1 2 2 0 1 −1 −1 0 0 1 −1 2 1 −1 1 2	Case #1: No Case #2: Yes

【题解】

难度等级:中等题,几何。FB 时间是 38 分钟。铜牌队伍在 220 分钟左右 AC。

能力考核:逻辑思维、较强的编码能力。

本题代码较长,逻辑较复杂,但是很容易理解,也不涉及复杂的算法。这种题是考查编码能力的典型题目,在编码时需要认真处理几何模板、逻辑关系等。

本题的代码已经在 11.2.1 节中作为例题详细给出。

12.2.5 B 题 Binary Tree(hdu 5573)

Problem Description:

青蛙国王住在一个无限长的树的根部。根据法律,每个结点应该连接下一层的两个结点,构成一个完整的二叉树。

因为国王的数学很牛,它为每个结点配置了一个数字。特别地,树的根就是国王住的地方,是 1,记为 froot=1。对每个结点 u,标签是 fu,左子结点是 fu×2,右子结点是 fu×2+1。国王对它的树王国很满意。

时间流逝,国王病了。根据黑魔法,如果国王能收集 N 个幽灵宝石,可以让它再活 N 年。开始时,国王位于根部,幽灵宝石的数量是 0,然后它往下走,每次往左子结点或右子结点走。在结点 x 处,这个结点的数字是 f_x(记住 froot=1),它可以选择把幽灵宝石增加 f_x 或者减少 f_x。它从根部开始走,访问 K 个 node(包括根结点),在每个结点处加或者减去上述的数字。如果数字最后是 N,它就成功了。注意幽灵宝石有魔法,幽灵宝石的数字 N 可能是负数。

给定 n、K,帮助国王收集 n 个幽灵宝石,不多不少访问 K 个结点。

Input:

第 1 行是一个整数 T,表示测试用例的个数。

每个测试用例包括两个整数 n 和 K,即国王要收集的幽灵宝石的数量、访问的结点数量。

Output:

对每个测试用例,先输出"Case #x:",其中 x 表示第几个用例,从 1 开始计数。

下面有 K 行,每一行'a b':a 是青蛙访问的结点标签;b 是'+'或'-',表示加或减 a。

保证至少有一个结果成立,如果有很多成立,可以输出任何一个。

Limits:

$1 \leqslant T \leqslant 100$;

$1 \leqslant n \leqslant 10^9$;

$n \leqslant 2^K \leqslant 2^{60}$。

Sample Input	Sample Output
2 5 3 10 4	Case #1: 1 + 3 - 7 + Case #2: 1 + 3 + 6 - 12 +

【题解】

难度等级：中等题，模拟题，构造。FB 时间是 44 分钟。铜牌队伍在 213 分钟左右 AC。

能力考核：计算思维、逻辑思维。

本题不需要复杂的算法，编码也不长，但是需要灵活处理，找到间接的解决方案。这是典型的考查思维能力的题目，具体是考查对二进制的领悟能力。这种思维能力需要聪明的头脑和长期的编程训练。

首先考虑计算复杂度。本题如果用暴力的方法，简单地罗列所有可能的走法是不行的。从第 1 层根结点走到第 K 层，一共有 2^{K-1} 个路径，由于每个结点上可取正负，那么每个路径上又有 2^K 种组合，合起来大概是 4^K，而 $K \leqslant 60$，肯定会 TLE。

类似这种题目，"灵机一动"非常重要。比如本题的思路是只要一直沿着最左边（加上第 K 层最左边的右子结点）走到第 K 层，就能找到一个答案。知道了这一点，后面就简单了。虽然题目中的限制条件 $n \leqslant 2^K$ 给出了暗示，但是想到这一点仍然很难。这也是为什么平时做题训练时尽量不要看题解，而是应该多自己思考，锻炼思维能力。如果靠看别人的题解知道这个方法，然后再去完成编码，收获是很小的。

上述思路证明如下：

（1）二叉树最左边的那条边，从上到下相加，满足条件 $n \leqslant 2^K$。从第 1 层沿着最左边走到第 K 层（图 12.1 中路径是 1-2-4-8），最大值是 $n = 2^0 + 2^1 + 2^2 + 2^3 + 2^4 + \cdots + 2^{K-1} = 2^K - 1$，如果在第 K 层选择最左边的右子结点（图 12.1 中路径是 1-2-4-9），那么最大值是 $N = 2^K$。

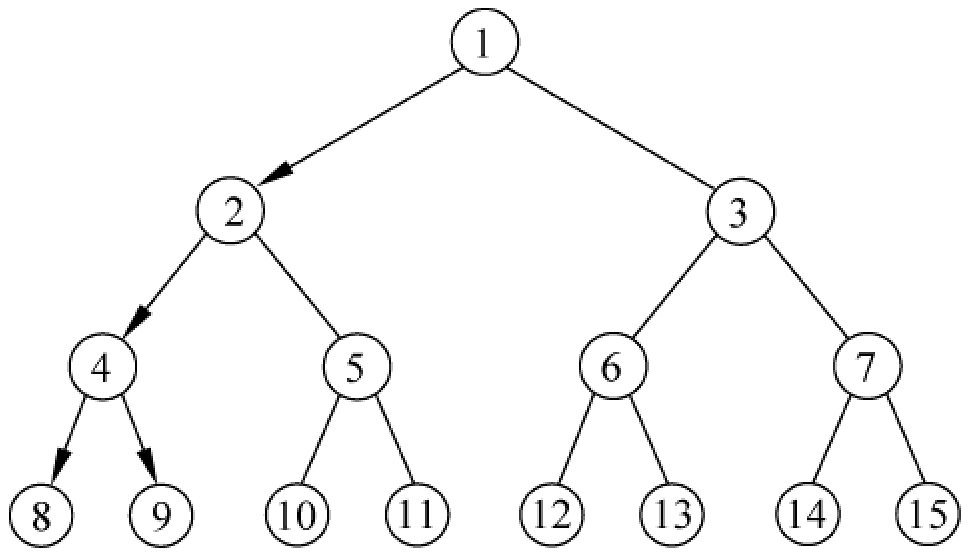


图 12.1　B 题

（2）对于给定的 K，沿着最左边走（最后一层可以走右边）可以实现 $1 \leqslant n \leqslant 2^K$ 中的所有值，也就是对任何 n 都能找到一个答案。请自己证明。下面用一个例子来说明：令 $K=5$，最左边的数依次是 1，2，4，8，16。最大 $n=32$，现在证明 1～32 中所有的数都能用 1，2，4，8，16（最后一层可以走右边）的组合获得。实际上，一直走左边可以得到奇数，然后在最后一层走右边能得到偶数，所以只需要考虑奇数就行了。

$31=16+8+4+2+1$。联想到 31 的二进制表示：$31=11111_2$；用 11111 表示这条路径，其中 1 表示'+'，0 表示'−'。

$29=16+8+4+2-1$，$29=31-2$，把上面 $16+8+4+2+1$ 中的 $+1$ 变成 -1 即可。用 11110 表示这条路径。11110 可以这样计算得到：$31-(31-29)/2=30=11110_2$。

$27=16+8+4-2+1$，$27=31-4$，把 $+2$ 变成 -2，2 的二进制是 10_2。用 11101 表示这条路径，$31-(31-27)/2=29=11101_2$。

25＝16＋8＋4－2－1，25＝31－6，把＋3 变成－3，3 的二进制是 11_2。用 11100 表示这条路径，31－(31－25)/2＝28＝11100_2。

23＝16＋8－4＋2＋1，23＝31－8，把＋4 变成－4；用 11011 表示这条路径。

21＝16＋8－4＋2－1，21＝31－10，把＋5 变成－5；用 11010 表示这条路径。

……

每个奇数都能实现。

在编程时，结合二进制的特点，很容易写出程序。其复杂度为 $O(K)$。

```
#include <algorithm>
typedef long long LL;
int main(){
    int num, n, K, odd;
    scanf("%d", &num);
    for(int i = 1; i <= num; ++i){
        scanf("%d%d", &n, &K);
    if(n % 2)
            odd = 0;              //n 是奇数,每一层都是最左边的数
        else {
            odd = 1;              //n 是偶数,转换为比它小 1 的奇数处理。最后一层取右边的数
            n--;
        }
        LL pp = (LL)pow(2,K) - 1;
            //二进制数为全 1 的数。例如 K = 5 时,pp = 31,二进制是 11111
        LL kk = pp - (pp - n)/2;  //kk 的二进制表示,就是一个可行的路径。
                                  //其中为 1 的是'+',为 0 的是'-'。原因见上文的证明
        LL pos = 0;               //当前层数,从国王的顶层开始
        printf("Case #%d:\n", i);
        while(kk > 1) {           //不处理最后一层,最后一层有奇偶问题
          if(kk & 1)              //二进制数的个位数是当前的层
                                  //这个位置是'1',表示要加
            printf("%lld %c\n", (LL)pow(2,pos), '+');
        else                      //二进制数的个位数是'0',表示要减
            printf("%lld %c\n", (LL)pow(2,pos), '-');
        kk = kk >> 1;             //二进制数右移一次,把处理过的移走
                                  //新的个位数是下一层
        pos++;
      }
//下面处理最后一层。如果 n 是偶数,最后一层取右边
      if(kk & 1)
          printf("%lld %c\n", (LL)pow(2,pos) + odd, '+');
      else
          printf("%lld %c\n", (LL)pow(2,pos) + odd, '-');
    }
    return 0;
}
```

12.2.6 D 题 Discover Water Tank(hdu 5575)

Problem Description：

水箱里面住着很多青蛙，但是它们都不知道水箱里有多少水。

水箱的高度无限，但是底部狭窄。水箱底部长 N，宽只有 1。

$N-1$ 个板子把水箱隔成 N 部分，每部分底部大小是 1×1，板子的高度不同。水不能穿过板子，但是如果水平面比板子高，根据基本的物理规律，水会从板子上面漫过去。

青蛙国王想知道水箱的细节，它派人选择了 M 个点，看这些点上有没有水。

例如，每次它选择(x,y)，表示在水箱的第 x 部分$(1\leqslant x\leqslant N$，从左到右计数)，在高度$(y+0.5)$的地方检查是否有水。

国王得到了 M 个结果，但是它发现有些可能是错的。国王想知道正确结果的最大可能个数有多少。

Input：

第 1 行是一个整数 T，表示测试用例的个数。

每个测试用例的第 1 行是两个数 N 和 M，即水箱隔成 N 部分、测量 M 次。

每个测试用例的第 2 行包括 $N-1$ 个整数，即 $h_1,h_2,\cdots,h_{N-1}$，h_i 表示第 i 个板子的高度。

下面有 M 行，第 i 行的格式为'x y z'，表示测量结果。如果第 x 个水箱高$(y+0.5)$处没有水，那么 $z=0$，否则 $z=1$。

Output：

对每个测试用例，需要输出"Case #x：y"，其中 x 表示第几个用例，从 1 开始计数；y 是正确结果的最大可能数字。

Limits：

$1\leqslant T\leqslant 100$；

90%的数据，$1\leqslant N\leqslant 1000$，$1\leqslant M\leqslant 2000$；

100%的数据，$1\leqslant N\leqslant 10^5$，$1\leqslant M\leqslant 2\times10^5$；

$1\leqslant h_i\leqslant 10^9$，$1\leqslant i\leqslant N-1$；

对每个结果，$1\leqslant x\leqslant N$，$1\leqslant y\leqslant 10^9$，$z$ 是 0 或 1。

Sample Input	Sample Output
2 3 4 3 4 1 3 1 2 1 0 2 2 0 3 3 1 2 2 2 1 2 0 1 2 1	Case #1：3 Case #2：1

【题解】

难度等级：难题，综合。FB 时间是 119 分钟。金牌队伍在 240 分钟左右 AC，银牌有少

数队伍做出。

能力考核：高级数据结构(左偏树)、STL 库、逻辑思维、编码能力。

本题有复杂的逻辑，大规模的数据，需要结合数据结构、算法、STL 库，是典型的难题，综合考查逻辑、算法、编码等多方面的能力。

(1) 计算复杂度。读题时首先应注意到本题较大的数据规模，即板子的数量 $1 \leqslant N \leqslant 10^5$，以及测量的个数 $1 \leqslant M \leqslant 2 \times 10^5$，因此计算复杂度比 $O(NM)$ 小。

(2) 理解题目。

图 12.2 是第 1 个测试样例。图中'×'表示无水，即 $z=0$；'O'表示有水，即 $z=1$。在这个例子中，最大的正确数字是 3，即第 1 个水箱的测量结果是错的，后面 3 个结果都是对的。

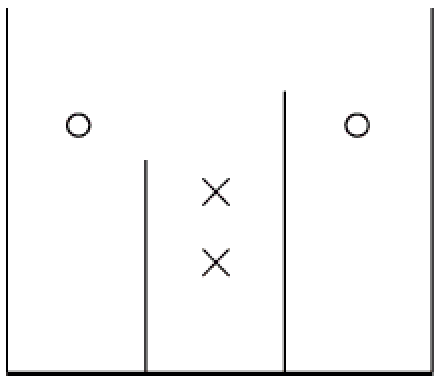


图 12.2　D 题

本题的解决思路是比较清晰的，即从低到高，逐渐给水箱加水，然后计算正确的测量个数。步骤如下：

① 从假设水箱没有水开始，此时'×'的个数就是答案，记为 ans。

② 逐一检查每个'O'的有水记录，即给这个水箱加水，直到这个记录的高度。可以想到，按从低到高的顺序检查所有的'O'，逻辑上是正确的，而且比较简单。

③ 上一步中对每个'O'的检查，加水到'O'的高度后，它可能向左、向右溢出。检查被它溢出的水箱，在当前水位高度下有多少'×'？有多少'O'？'O'的数量减去'×'的数量，差值为 d，更新 ans，ans=ans+d。

④ 检查完所有'O'，输出 ans。

在以上思路中，③是最关键的。检查每个'O'向左、向右的溢出，复杂度是 $O(N)$；一共有 M 个'O'，总复杂度是 $O(MN)$。而 $1 \leqslant N \leqslant 10^5$，$1 \leqslant M \leqslant 2 \times 10^5$，$O(MN)$很大，显然不能用暴力的方法去检查每个'O'的溢出。那么怎么做呢？因为是按从低到高的顺序检查'O'，在检查更高的'O'时，前面检查过的、相邻水箱的、较低的'O'的检查结果可以直接拿来用。或者说，一些相邻的水箱可以合并为一个大水箱。

当水向左、向右溢出以后，被溢出到的小水箱合并成一个大水箱，这样 $O(M)$次枚举'O'的复杂度就是 $O(N)$，而不是 $O(MN)$了。

在合并的时候，不仅要合并当前水位之下的'O'和'×'的数量，还要维护大水箱的左、右两边是哪些水箱以及是哪些挡板。'O'的数量可以看当前枚举了多少 O，'×'的数量可以用一个能表示顺序的数据结构来维护，这个数据结构维护结构体(x,y)表示'×'在坐标为(x,y)的位置。在计算第 x 个小水箱的水位 h 下的'×'的时候，可以在删除数据结构最小元素的同时计数，直到 $y \geqslant h$(即最低位置的'×'在水位之上)为止。维护顺序的数据结构可以用堆或者平衡树，但是又需要数据能够快速地合并，因此使用由左偏树实现的可并堆。

概括起来，解题过程是以 $z=0$ 的测量总数为初始答案，通过枚举 $z=1$ 的情况来更新答案，维护水箱并处理水箱的合并，用左偏树实现的可并堆来处理水箱的合并及计算单次枚举的答案。总的复杂度是 $O(N\log N+N\log M)$。

下面是出题人提供的代码。

```
#include<bits/stdc++.h>
using namespace std;
const int N = 200100;
```

```
const int INF = 2000000000;
int n, m, h[N], st[N], wh[N], height[N * 2], val[N * 2][2], dp[N * 2][2];
int lca[N * 2][18], dep[N * 2];
vector<int> pos[2], off[2], high[N * 2][2];
vector<int> edge[2 * N];
void clear(){
    for(int i = 0; i <= n; ++i) {
        edge[i].clear();
        high[i][0].clear(); high[i][1].clear();
    }
    pos[0].clear(); off[0].clear();
    pos[1].clear(); off[1].clear();
}
void pre_dfs(int u, int fa, int dist) {
    lca[u][0] = fa; dep[u] = dist;
    for(int i = 1; i < 18; ++i)
        lca[u][i] = lca[lca[u][i - 1]][i - 1];
    for(int i = 0; i < edge[u].size(); ++i) {
        int v = edge[u][i];
        pre_dfs(v, u, dist + 1);
    }
}
int get_node(int x, int y) {
    for(int i = 17; i >= 0; --i)
        if(dep[x] > (1 << i) && height[lca[x][i]] <= y)
            x = lca[x][i];
    return x;
}
void add_edge(int x, int y) {
    edge[x].push_back(y);
}
void build_tree() {
    int tot = n, top = 0;
    h[n] = INF;height[0] = INF;
    for(int i = 1; i <= n; ++i) {
        height[i] = 0; val[i][0] = val[i][1] = 0;
        wh[top] = i;
        while(top > 0 && st[top - 1] < h[i]) {
            ++tot;
            height[tot] = st[top - 1];
            val[tot][0] = val[tot][1] = 0;
            int tid = top - 1;
            add_edge(tot, wh[top]);
            while(top > 0 && st[top - 1] == st[tid]){
                --top;
                add_edge(tot, wh[top]);
            }
            wh[top] = tot;
        }
        st[top++] = h[i];
    }
```

```
    n = tot;
    for(int i = 0; i < 18; ++ i)
        lca[0][i] = 0;
    pre_dfs(n, 0, 1);
    for(int i = 0; i < 2; ++ i) {
        for(int j = 0; j < pos[i].size(); ++ j) {
            int id = get_node(pos[i][j], off[i][j]);
            val[id][i] ++; high[id][i].push_back(off[i][j]);
        }
    }
}
void dfs(int u) {
    dp[u][0] = val[u][0];
    dp[u][1] = val[u][1];
    sort(high[u][0].begin(), high[u][0].end());
    sort(high[u][1].begin(), high[u][1].end());
    int p = 0, ret = 0;
    for(int i = 0; i < val[u][0]; ++ i) {
        while(p < val[u][1] && high[u][1][p] < high[u][0][i])
            ++ p;
        ret = max(ret, p + val[u][0] - i);
    }
    int flag = val[u][0], Mind = INF;
    for(int i = 0; i < edge[u].size(); ++ i) {
        int v = edge[u][i];
        dfs(v);
        dp[u][1] += dp[v][1];
        dp[u][0] += max(dp[v][0], dp[v][1]);
    }
    dp[u][0] = max(dp[u][0], dp[u][1] - val[u][1] + ret);
}
void solve() {
    scanf("%d%d", &n, &m);
    for(int i = 1; i < n; ++ i)
        scanf("%d", h + i);
    for(int i = 1; i <= m; ++ i) {
        int x, y, z;
        scanf("%d%d%d", &x, &y, &z);
        pos[z].push_back(x); off[z].push_back(y);
    }
    build_tree();
    dfs(n);
    printf("%d\n", max(dp[n][0], dp[n][1]));
    clear();
}
int main() {
    int t;
    scanf("%d", &t);
    for(int i = 1; i <= t; ++ i) {
        printf("Case #%d: ", i);
        solve();
```

```
    }
    return 0;
}
```

12.2.7　E 题 Expection of String(hdu 5576)

Problem Description:

青蛙刚学会了乘法,现在它想做一些练习。

它在纸上写了一个字符串,只包括数字和一个'×'符号。如果'×'出现在字符串前面或后面,它认为结果是0,否则它将按正常乘法来计算。

在练习之后,它想到一个新问题:对一个初始字符串,每次随机选两个字符交换位置,它交换了一次又一次,例如 K 次,它想知道新字符串计算结果的期望值。

可以知道所有交换的方法一共有 $\binom{n}{2}^K$ 种(即$(C_n^2)^K$)。如果期望的结果是 x,需要输出整数 $x\times\binom{n}{2}^K$。

Input:

第1行是一个整数 T,表示测试用例的个数。

每个测试用例的第1行是一个数 K,表示青蛙交换字符的次数。

每个测试用例的第2行是青蛙操作的字符串,只包括数字和一个乘法操作符'*'。

Output:

对每个测试用例,需要先输出"Case #x: y",其中 x 表示第几个用例,从1开始计数;y 是结果。

由于 y 可能很大,用 10^9+7 取模。

Limits:

$1\leqslant T\leqslant 100$;

字符串长度为 L;

70%的数据,$1\leqslant L\leqslant 10$,$0\leqslant K\leqslant 5$;

95%的数据,$1\leqslant L\leqslant 20$,$0\leqslant K\leqslant 20$;

100%的数据,$1\leqslant L\leqslant 50$,$0\leqslant K\leqslant 50$。

Sample Input	Sample Output
2 1 1*2 2 1*2	Case #1: 2 Case #2: 6

【题解】

难度等级:难题。FB 时间是 118 分钟。部分金牌队伍做出,AC 时间在 230 分钟左右。

能力考核:DP、逻辑思维、编码能力。

本题的逻辑很复杂，数据大，是典型的难题，综合考查逻辑、算法、编码等多方面的能力。

题目的要求是一个字符串由数字和'×'号组成，每次操作可以交换其中任意两个符号(包括'×')，问 K 次操作后所有可能结果的和。

样例1，字符串"1 * 2"，交换一次，结果有 $(C_n^2)^K=(C_3^2)^1=3$ 种情况，即 * 12、2 * 1、12 * 。期望值 $x=\frac{0}{3}+\frac{2}{3}+\frac{0}{3}=\frac{2}{3}$，输出 $x\times(C_n^2)^K=\frac{2}{3}\times3=2$。

样例2，字符串"1 * 2"，交换两次，结果可能有 $(C_n^2)^K=(C_3^2)^2=9$ 种，分别是 1 * 2、21 * 、* 21、* 21、1 * 2、21 * 、21 * 、* 21、1 * 2，如图 12.3 所示。期望值 $x=\frac{2}{9}+\frac{2}{9}+\frac{2}{9}=\frac{6}{9}$，输出 $x\times(C_n^2)^K=\frac{6}{9}\times9=6$。

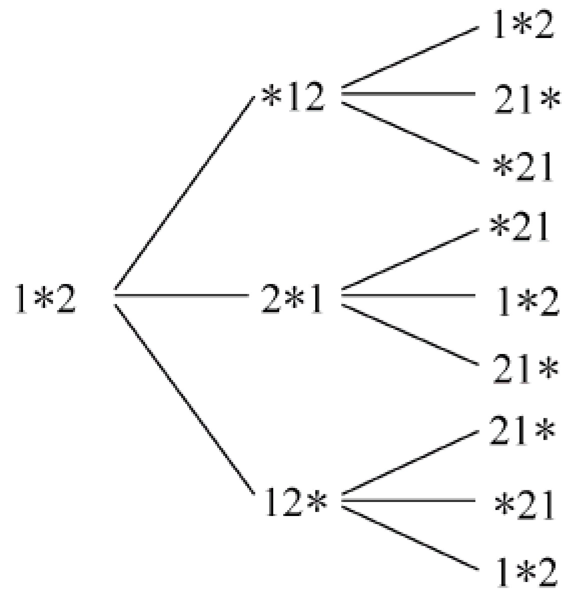


图 12.3　E 题

本题如果用暴力的方法，逐一检查每个结果，可能有 $(C_n^2)^K\leqslant(C_{50}^2)^{50}$ 种情况，数据太大，不可能进行计算。

用 DP 实现，关键是递推式，复杂度为 $O(Kn^3)$。

下面是出题人提供的代码。

```
#include <bits/stdc++.h>
using namespace std;
#define ll long long
int K,n;
const ll mod = 1000000007;
ll dp[2][55][55][55];
ll ten[55],tot[55],tot_2[55][55];
string str;
int main(){
    int T;
    int cas = 1;
    cin >> T;
    ten[0] = ten[1] = 1;
    for (int i = 2; i <= 50; i++)
        ten[i] = ten[i - 1] * 10 % mod;
    while (T -- ) {
        cin >> K >> str;
        n = str.size();
        int now = 0;
        for (int i = 0; i < n; i++)
        if (str[i] == '*') {
            for (int j = 0; j < n; j++)
                for (int k = j + 1; k < n; k++)
                if (str[j] == '*'|| str[k] == '*')
                    dp[now][i][j][k] = 0;
                else
                    dp[now][i][j][k] = (str[j] - '0') * (str[k] - '0');
        }
        else {
            for (int j = 0; j < n; j++)
```

```
            for (int k = 0; k < n; k++)
                dp[now][i][j][k] = 0;
}
ll lamb;
if (n <= 3) lamb = 1;
else lamb = (n - 3) * (n - 4) / 2 + 1;
for (int iter = 0; iter < K; iter++) {
    now = 1 - now;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                dp[now][i][j][k] = 0;
    for (int j = 0; j < n; j++)
        for (int k = j + 1; k < n; k++) {
            tot_2[j][k] = 0;
            for (int i = 0; i < n; i++)
            if (i != j && i != k)
                tot_2[j][k] += dp[1 - now][i][j][k];
            tot_2[j][k] % = mod;
        }
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
        if (i != j) {
            tot[j] = 0;
            for (int k = 0; k < n; k++)
            if (k != i && k != j)
                tot[j] += dp[1 - now][i][min(j,k)][max(j,k)];
            tot[j] % = mod;
        }
        for (int j = 0; j < n; j++)
            for (int k = j + 1; k < n; k++)
            if (j != i && k != i) {
                dp[now][i][j][k] += dp[1 - now][i][j][k] * lamb;
                dp[now][i][j][k] += tot[j] - dp[1 - now][i][j][k] + mod;
                dp[now][i][j][k] += tot[k] - dp[1 - now][i][j][k] + mod;
                dp[now][i][j][k] += tot_2[j][k] - dp[1 - now][i][j][k] + mod;
                dp[now][i][j][k] += dp[1 - now][j][min(i,k)][max(i,k)]
                                  + dp[1 - now][k][min(i,j)][max(i,j)];
            }
    }
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++) {
                dp[now][i][j][k] % = mod;
            }
}
ll ans = 0;
for (int i = 1; i < n - 1; i++)
    for (int j = 0; j < i; j++)
        for (int k = i + 1; k < n; k++)
            ans += dp[now][i][j][k] * ten[i - j] % mod * ten[n - k] % mod;
```

```
        printf("Case # %d: %lld\n", cas++, ans % mod);
    }
}
```

12.2.8 G 题 Game of Arrays(hdu 5579)

Problem Description:

Tweek 和 Craig 是好朋友,总在一起玩。在做数学作业的时候,他们发明了一种新游戏。

首先,他们写了 3 个数组 A、B、C,每个有 N 个数字。接着,他们在黑板上写下:

$$A+B=C$$

如果等式满足,说明从 1 到 N 的所有位置 $A_i+B_i=C_i$ 都成立。

当然,开始的时候等式不是都成立。

很幸运,数组 A、B、C 的一些数字可以改变,一些不能改。那些可以改的数字的位置在游戏前固定好了。

在游戏中,Tweek 先走,然后两人轮流进行,每次可以改变一个数字。

每一次,游戏者可以从一个数组中选一个可改变的数字,减去 1。但是,不能出现负数,所以被选中的数字在做减法前不能是 0。

Tweek 的目标是在游戏中使等式成立,而 Craig 的目标是阻止。

当等式成立时游戏结束,Tweek 获胜。或者不存在可能的改变,$A+B\neq C$(至少有一个 $i\in[1,N]$,使得 $A_i+B_i\neq C_i$),Craig 获胜。

给定 A、B、C,以及每个数组可改变数字的位置。本题的任务是确定谁获胜。

Input:

第 1 行是一个整数 T,表示测试用例的个数。

每个测试用例的第 1 行是一个整数 K,表示数组 A、B、C 的长度。

每个测试用例的第 2 行和第 3 行描述数组 A。第 2 行包括 N 个整数 $A_1,A_2,\cdots,A_N$,表示数组 A 的元素。第 3 行包括 N 个整数 $u_1,u_2,\cdots,u_N$,如果 A_i 可改,u_i 是 1,否则 u_i 是 0。

每个测试用例的第 4 行和第 5 行描述数组 B。第 4 行包括 N 个整数 $B_1,B_2,\cdots,B_N$,表示数组 B 的元素。第 5 行包括 N 个整数 $v_1,v_2,\cdots,v_N$,如果 B_i 可改,v_i 是 1,否则 v_i 是 0。

每个测试用例的第 6 行和第 7 行描述数组 C。第 6 行包括 N 个整数 $C_1,C_2,\cdots,C_N$,表示数组 C 的元素。第 7 行包括 N 个整数 $w_1,w_2,\cdots,w_N$,如果 C_i 可改,w_i 是 1,否则 w_i 是 0。

Output:

对每个测试用例,需要先输出"Case #x: y",其中 x 表示第几个用例,从 1 开始计数;y 是获胜者。

Limits:

$1\leqslant T\leqslant 2000$;

75%的数据,$1\leqslant N\leqslant 10$;

95%的数据,$1\leqslant N\leqslant 50$;

100%的数据，$1\leqslant N\leqslant 100$；

$0\leqslant A_i, B_i, C_i\leqslant 10^9$；

u_i、v_i、w_i 值为 0 或 1。

Sample Input	Sample Output
3 2 4 3 1 1 4 4 0 1 5 5 0 0 2 4 4 1 1 4 4 0 0 5 5 0 0 2 4 4 1 1 4 4 0 0 4 4 0 0	Case #1: Tweek Case #2: Craig Case #3: Tweek

【题解】

难度等级：本题是“难题”。FB 时间是 188 分钟。部分金牌队伍做出，AC 时间在 270 分钟左右。

能力考核：数学、逻辑思维、编码能力。

本题综合考查逻辑、算法、编码等多方面的能力。

下面是出题人提供的代码。

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1100;
int a[N], b[N], c[N], ta[N], tb[N], tc[N], n, d[N], add[N], del[N];
bool check() {
    int tot = 0;
    for(int i = 1; i <= n; ++ i) {
        d[i] = a[i] + b[i] - c[i];
        add[i] = a[i] * ta[i] + b[i] * tb[i];
        del[i] = c[i] * tc[i];
```

```
            tot += abs(d[i]);
        }
        for(int i = 1; i <= n; ++ i) {
            if((d[i] > 0 && d[i] > add[i] - del[i]) ||
                    (d[i] < 0 && - d[i] > del[i] - add[i]))
                return 0;
            if((d[i] >= 0 && d[i] < add[i] - del[i]) ||
                    (d[i] <= 0 && - d[i] < del[i] - add[i])) {
                if(tot - abs(d[i]) > abs(d[i]))
                    return 0;
            }
        }
        return 1;
    }
    void solve(int cas) {
        scanf("%d", &n);
        for(int i = 1; i <= n; ++ i) scanf("%d", a + i);
        for(int i = 1; i <= n; ++ i) scanf("%d", ta + i);
        for(int i = 1; i <= n; ++ i) scanf("%d", b + i);
        for(int i = 1; i <= n; ++ i) scanf("%d", tb + i);
        for(int i = 1; i <= n; ++ i) scanf("%d", c + i);
        for(int i = 1; i <= n; ++ i) scanf("%d", tc + i);
        int win = 1;
        for(int i = 1; i <= n; ++ i) {
            if(a[i] + b[i] != c[i]) {
                win = 0;
                break;
            }
        }
        if(win){puts("Tweek"); return;}
        for(int i = 1; i <= n; ++i){
            if(ta[i] && a[i] > 0){
                a[i] --;
                if(check()){puts("Tweek");return;}
                a[i] ++;
            }
            if(tb[i] && b[i] > 0) {
                b[i] --;
                if(check()){puts("Tweek");return;}
                b[i] ++;
            }
            if(tc[i] && c[i] > 0) {
                c[i] --;
                if(check()){puts("Tweek");return;}
                c[i] ++;
            }
        }
        puts("Craig");
```

```
        return;
    }
    int main(){
        int t;
        scanf(" % d", &t);
        for(int i = 1; i <= t; ++ i) {
            printf("Case # % d: ", i);
            solve(i);
        }
        return 0;
    }
```

12.2.9　I 题 Infinity Point Sets(hdu 5581)

Problem Description:

这个故事来自一位古老的青蛙哲学家写的古书。

什么时候会是世界的尽头？也许最好的理解方法是使用几何进行计算。

起初,应该在一张纸上画几个点。每次选择两个点并用线段连接起来。当有两个线段交叉时,在交叉点上产生一个新点,将该点添加到纸上,并尝试像之前一样将其与前面的点连接。

应该一次又一次地执行此操作,继续绘制线段,并在可能的情况下添加点,直到没有新线段为止。然后是世界的尽头,旧的青蛙会死亡,新的时代将开始。

如你所见,不同的初始点导致不同的结果。对于一些点集合,世界的末日永远不会到来,我们称之为无穷大集合。

现在给 N 个点,在这 N 个点的所有可能子集(不包括空集,所以总共将有 2^N-1 个集)中有多少个不是无穷大?

Input:

第 1 行是一个整数 T,表示测试用例的个数。

每个测试用例都以整数 N 开始,它表示点的数量。

在下面的 N 行中,第 i 行包含两个整数 x_i 和 y_i,表示第 i 点的坐标(x_i,y_i)。

Output:

对每个测试用例,需要输出"Case # x: y",其中 x 表示第几个用例,从 1 开始计数; y 是结果。

由于 y 可能很大,用 10^9+7 取模。

Limits:

$1 \leqslant T \leqslant 10$;

90%的数据,$1 \leqslant N \leqslant 100$;

100%的数据,$1 \leqslant N \leqslant 1000$;

$1 \leqslant x_i, y_i \leqslant 10^4$;

没有一对具有相同坐标的点。

Sample Input	Sample Output
2 4 0 0 0 2 2 2 2 0 5 0 0 0 2 2 2 2 0 1 2	Case ＃1：15 Case ＃2：30

【题解】

难度等级：本题是“难题”。FB 时间是 197 分钟。部分金牌队伍做出。

能力考核：几何、逻辑思维、编码能力。

本题综合考查逻辑、算法、编码等多方面的能力。题目大意是给出二维空间里 n 个点的坐标，求有多少个不同的子点集不是无限点集。无限点集的定义是将点集中的点两两相连，将线段产生的交点再加入点集中，继续上面的操作，如果操作能够无限地进行下去，则称之为无限点集。

图 12.4 所示的 4 种情况不是无限点集。

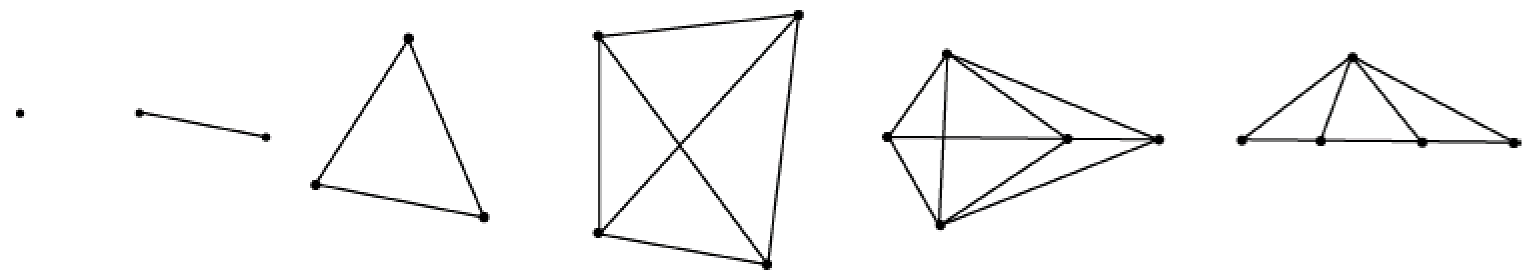


图 12.4　I 题

(1) 任意 1、2、3、4 个点。

(2) 3 点以上共线＋两侧各一点。生成的新的黑色的点也在线段上，操作不会无限地进行。

(3) 4 点以上共线＋任意一点。这种情况不会产生交点，操作也不会无限进行。

(4) 5 点及以上共线。

无限点集的情况例如图 12.5 所示。5 个外面的点，交点为 5 个内部的点，这样的操作能够无限地进行下去。

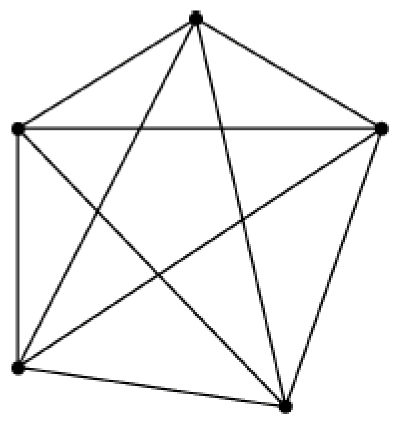


图 12.5　5 个点的情况

下面是出题人提供的代码。

```
#include <bits/stdc++.h>
typedef long long LL;
using namespace std;
const int V = 1100;
```

```
const int N = 1000;
const int P = 1000000007;
int rev[V], pt[V], C[V][V];
int Pow(int x, int y){
    int ret = 1;
    while(y){
        if(y & 1) ret =  (LL) ret * x % P;
        x = (LL) x * x % P;
        y /= 2;
    }
    return ret;
}
void init(){
    for(int i = 1; i <= N; ++i)
        rev[i] = Pow(i, P - 2);
    pt[0] = 1;
    for(int i = 1; i <= N; ++i)
        pt[i] = pt[i - 1] * 2 % P;
    memset(C, 0, sizeof(C));
    for(int i = 0; i <= N; ++i){
        C[i][0] = C[i][i] = 1;
        for(int j = 1; j < i; ++j)
            C[i][j] = (C[i - 1][j - 1] + C[i - 1][j]) % P;
    }
}
struct Point{
    int x, y;
}p[V];
struct PNode{
    int x, y, rev;
}Node[V];
bool EQ(PNode x, PNode y){
    if(x.x == y.x && x.x == 0) return true;
    if(x.x * y.x < 0) return false;
    return x.x * y.y == x.y * y.x;
}
bool Nodecmp(PNode x, PNode y){
    if(x.x * y.x <= 0) return x.x > y.x;
    if(x.x * y.y != x.y * y.x){
        if(x.x >= 0) return x.x * y.y > x.y * y.x;
        else return x.x * y.y > x.y * y.x;
    }
    return x.rev < y.rev;
}
int _, n;
/* 分为几部分: (1) 5 点及以上共线; (2) 任意 1、2、3、4 个点;
              (3) 4 点以上共线 + 任意一点; (4) 3 点以上共线 + 两侧各一点 */
int sol_line(int ln, int rn, int revn, int nown, int total){
    int ans = 0;
    for(int i = 4; i <= ln + rn; ++i)
        ans = (ans + (LL)C[ln + rn][i] * rev[i + 1] % P) % P;
```

```
    for(int i = 3; i <= ln + rn; ++i)
        ans = (ans + (LL)C[ln + rn][i] * rev[i + 1] % P * (n - ln - rn - 1) % P) % P;
    int D = revn - nown;
    int A = revn - D - rn;
    int c = total - A - ln - rn;
    int CD = c + D;
    int AB = n - 1 - ln - rn - CD;
    for(int i = 2; i <= ln + rn; ++i)
        ans = (ans + (LL)C[ln + rn][i] * rev[i + 1] % P * AB % P * CD % P) % P;
    return ans;
}
int mid_way[V];
int sol(){
    int ret = 0;
    for(int i = 1; i <= 4; ++i)
    ret = (ret + C[n][i]) % P;
    for(int i = 0; i < n; ++i){
        int revn = 0;
        for(int j = 0; j < n; ++j){
            Node[j].x = p[j].x - p[i].x;
            Node[j].y = p[j].y - p[i].y;
            if(Node[j].y < 0 || (Node[j].y == 0 && Node[j].x < 0)){
                Node[j].x = -Node[j].x;
                Node[j].y = -Node[j].y;
                Node[j].rev = 1;
                ++revn;
            }
            else Node[j].rev = -1;
        }
        sort(Node, Node + n, Nodecmp);
        int ln = 0, rn = 0, midn = 0, nown = 0, total = 0, pre = -1;
        for(int j = 0; j < n; ++j){
            if(Node[j].x == 0 && Node[j].y == 0) continue;
            if(pre != -1 && !EQ(Node[j], Node[pre])){
                ret += sol_line(ln, rn, revn, nown, total);
                ret % = P;
                mid_way[midn++] = (LL) ln * rn % P;
                ln = rn = 0;
            }
            if(Node[j].rev == -1)  ++ln;
            else ++nown, ++rn;
            ++total;
            pre = j;
        }
        mid_way[midn++] = (LL) ln * rn % P;
        ret += sol_line(ln, rn, revn, nown, total);
        ret % = P;
        int mids = 0;
        for(int j = 0; j < midn; ++j) mids = (mids + mid_way[j]) % P;
        for(int j = 0; j < midn; ++j){
            ret = (ret - (LL)(mids - mid_way[j]) * mid_way[j] % P * rev[2] % P) % P;
```

```
            if(ret < 0) ret += P;
        }
    }
    return ret;
}
int main(){
    init();
    scanf("%d", &_);
    for(int ca = 1; ca <= _; ++ca){
        scanf("%d", &n);
        for(int i = 0; i < n; ++i)
        scanf("%d%d", &p[i].x, &p[i].y);
        printf("Case #%d: %d\n", ca, sol());
    }
    return 0;
}
```

参 考 文 献

[1] 刘汝佳，陈锋. 算法竞赛入门经典训练指南[M]. 北京：清华大学出版社，2012.
[2] 刘汝佳. 算法竞赛入门经典[M]. 2 版. 北京：清华大学出版社，2014.
[3] 余立功. ACM/ICPC 算法训练教程[M]. 北京：清华大学出版社，2013.
[4] 秋叶拓哉，岩田阳一，北川宜稔. 挑战程序设计竞赛[M]. 巫泽俊，等译. 2 版. 北京：人民邮电出版社，2013.
[5] 金博，郭立，于瑞云. 计算几何及应用[M]. 哈尔滨：哈尔滨工业大学出版社，2012.
[6] 俞勇. ACM 国际大学生程序设计竞赛算法与实现[M]. 北京：清华大学出版社，2013.
[7] Levitin A. 算法设计与分析基础[M]. 潘彦，译. 2 版. 北京：清华大学出版社，2007.
[8] Cormen T H，Leiserson C E. 算法导论[M]. 潘金贵，等译. 北京：机械工业出版社，2006.